CMU-CS-82-117

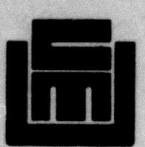
(12)

A Language-Oriented Interactive Programming Environment Based on Compilation Technology

Peter H. Feiler

May 1982

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

A Language-Oriented Interactive Programming Environment Based on Compilation Technology

Peter H. Feiler

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

May 1982

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was sponsored in part by the Software Engineering Division of CENTACS/CORADCOM, in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Abstract

This work is a feasibility study of a Language-Oriented Interactive Programming Environment (LOIPE) based solely on compilation. In this new approach to compiler-based environments we show that integrated interactive programming environments are not limited to interpretive systems, and that programming environments can have an understanding of the programmer's task and actively contribute to its solution. A prototype of LOIPE has been implemented.

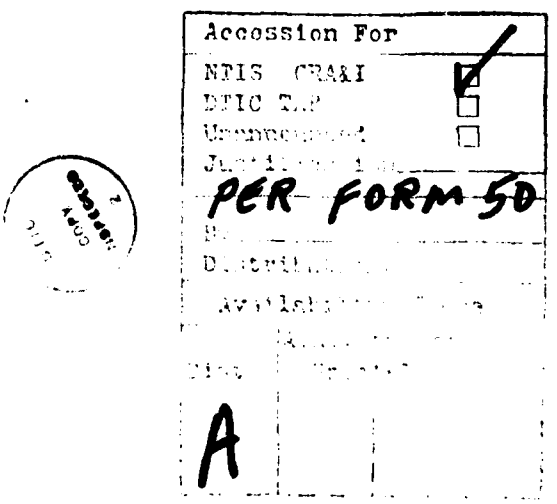
LOIPE is *language-oriented*. Both language-oriented program construction and language-oriented debugging are supported through a syntax-directed structure editor. This editor maintains a program tree as the primary program representation, but presents the user with a textual source program view. The debugger's functions are expressed in form of language constructs, utilizing the expressive power and abstraction mechanisms of the supported language, and are invoked by editing the program tree.

LOIPE is an *integrated* programming environment centered around the program tree. The structure editor is used as a uniform user interface. LOIPE takes responsibility for the integrity of the program data base, i.e., the program tree augmented with semantic and status information and the executable object code. The maintenance of the program representations and the invocation of tools is hidden from the user. The distinction between program manipulation and debugging diminishes.

LOIPE is *interactive*. Semantic errors are detected and reported while the user is still in context. Flexibility is provided by supporting the partial execution of programs with incomplete or semantically incorrect parts. Programs can be executed at any time.

LOIPE is an *incremental* programming environment. All processing steps of the compiler-based environment are performed incrementally between editing operations allowing for fast response time. Even LOIPE's debugger is realized with this incremental program replacement mechanism. By doing so debugging for optimizing code generators are supported.

We discuss the design and implementation of LOIPE both from the user's view and under the aspect of a solely compiler-based system. Our feasibility claim of LOIPE is substantiated by an evaluation of the design and the prototype implementation. The evaluation includes measurements on the running prototype and a discussion of the generation of LOIPE for a specific language.



Acknowledgements

This dissertation was developed under the guidance and patience of my advisor Nico Habermann. The thesis work contributed to the Gandalf project and benefitted from the many discussions with the members of the project -- Barb Denny, Bob Ellison, Dave Garlan, Gail Kaiser, Raul Medina-Mora, Dave Notkin, Dwayne Perry, Steve Popovich -- and especially from Raul Medina-Mora's thesis work on structure editors. The comments and suggestions from the other members of my thesis committee, Anita Jones, Walter Tichy, and Bill Wulf, improved the presentation of the thesis considerably.

I would like to acknowledge the departmental computers and the Scribe system that facilitated the production of this document. I am also grateful to Karl Zaininger and Horst Mauersberg for letting me finish the thesis while working for Siemens Corporation over the last year. Last, not least, immeasurable thanks to my wife Holly for being on my side in every way during my battle with the dissertation.

Table of Contents

List of Figures

Chapter 1

Introduction

The purpose of the research in this dissertation is to study a new approach in supporting a programmer with the construction and maintenance of programs. Our goal is to provide an integrated interactive programming environment that has some understanding of the programmer's tasks and actively participates in their solution. As a result it reduces the amount of time spent by a programmer at the possible expense of computer resources, which are readily available on the new generation of computers, the *personal computers*. This programming environment is known under the term *Language-Oriented Interactive Programming Environment* (LOIPE).

LOIPE is an integrated environment in which the editor, the compiler, the debugger, and the user interact through a single interface. The uniformity of the interaction is determined by the supported source language. The user no longer edits programs in terms of text but in terms of language constructs. The system contributes to the programming activity and informs the user of errors while the user is still in context. Similarly, debugging is performed at the language level and not in terms of the object code. All parts of the environment use a common data base for maintenance of the program. This *program data base* consists of the program source representation, a *program tree*, that is augmented with semantic and status information, and the executable object code representation. It is the responsibility of LOIPE to maintain the integrity of the program data base. Instead of the typical pipelining of application of software tools to the program text, LOIPE automatically invokes the various system parts incrementally in order to keep the program data base consistent. The executing program reflects the user's view of the source program at any time. Appendix B shows a user session with LOIPE.

The LOIPE contributes to the enhancement of program development in several ways:

- Uniformity — Integration provides the basis for uniformity. The user is presented with a source language view of both the program and the program execution. Other program representations are hidden. The user interacts with different parts of the environment in the same manner because they use a common interface. The transition between different programming activities is not noticeable. Uniformity extends to the implementation of LOIPE. An internal high-level program representation in form of a syntax tree is the primary representation that is common to all parts in the environment. Both the text form and the executable representation are generated from it. The use of a common program data base avoids redundancy of information and mechanisms.

- Active Participation — LOIPE takes responsibility for certain tasks in the programming process. It maintains an executable representation of the program that is consistent with the source program as seen by the user. All effects of a program modification are propagated by LOIPE such that all status information in the program data base is consistent. The complete program data base is automatically stored in the file system. The user is neither concerned with the application of the appropriate tool at the right time, nor with the use of the underlying file system.

- Language-Oriented Programming and Debugging — The notion of language-oriented structure manipulation (commonly known as syntax-directed editing) is used consistently as the means of communication between the programmer and the whole system. Both program construction and program debugging are performed by manipulating programs and their execution state in terms of their logical structure as defined by a high-level programming language, such as Pascal or Ada. As a result, the debugging facility provides functions that take advantage of the expressive power of the programming language and works at the level of abstractions in the source program.

- Responsive Behavior — All programmer activities have a predictable response time. This response time is relatively short, such that the programmer does not have to wait for the system to do its share of the work, no matter whether he wants to add another statement or switch to program execution. The notion of incremental update, being applied consistently in all system parts, contributes significantly to this goal.

- Flexibility — The user may manipulate the logical program structure in any way he desires. LOIPE detects semantic errors, but does not enforce their correction. In addition, the user may attempt to execute the program at any time. The program will execute until an incomplete or erroneous piece of the program is reached.

The dissertation is a feasibility study of an interactive programming environment with the above characteristics, whose implementation is based on *compilation* rather than interpretation. We demonstrate that flexibility and fast response of a programming environment do not require the use of the interpretive approach but can also be provided through use of compilation techniques. The LOIPE system combines the efficiency of

executing programs in compiler-based systems with flexibility in program manipulation uniformly through one mechanism, namely *incremental program replacement*. A benefit of this approach is that *remote program development*, i.e., the computer executing the program differs from the host computer on which the support system runs, can be supported with little effort.

We maintain that

- active participation of LOIPE simplifies the programmer's task by LOIPE taking responsibility for certain chores,
- our support system provides a framework for the generation of environments for different languages,
- and the system is expandable into a progam development system supporting multiple versions of programs and several programmers simultaneously.

We have implemented a demonstration system in order to substantiate the claim that our approach is feasible, i.e., can be done with satisfactory efficiency, and to test and refine the basic ideas.

In the remainder of this chapter we define the term programming environment for the context of the dissertation, discuss some of the important properties of LOIPE, review other work in the area of programming environments in relation to LOIPE. We also give a plan of the thesis.

1.1. A Definition of Programming Environment

There are many interpretations of the term programming environment. On one hand, a programming environment is sometimes understood to consist of a programming language, an editor, a compiler or interpreter, and a symbol debugger. On the other hand, it can have a very broad meaning. In an effort to standardize the support facilities for the programming language Ada, the DoD specified a programming support environment and gave guidelines for its implementation [DoD 78, Buxton 80]. The term programming support environment includes tools such as project budgeting, automatic report generation, and program libraries.

The Gandalf project [Habermann 79a] understands a programming environment to be a *Program Development and Maintenance Environment* for programming, system composition, and project coordination. Programming facilities support an individual programmer on a

program in relative isolation from others. System description facilities deal with problems that are related to the maintenance of programs, that exist in multiple versions, and are composed of smaller entities. Project coordination support addresses problems such as coordinating programmers and controlling the access of programmers to parts of the project, and maintaining consistent documentation.

In this dissertation we define a *programming environment* to be a support environment for a single programmer dealing with a single version of a program. The basic facilities of such a programming environment are

- an editor to enter and modify programs,
- a checker to ensure the syntax and the semantic correctness of the program,
- a translator to convert the program into an executable representation,
- a linker/loader to combine separately translated program pieces into one executable program and to place it into the computer memory for execution,
- a debugger for monitoring the execution of the program and for locating errors.

In addition to these basic facilities a programming environment for large programs must provide managerial support. This includes mechanisms for maintaining up-to-date copies of different program representations and for checking the interfaces of the building blocks of a program.

LOIPE is a programming environment of the above type with the following characteristics. The facilities for program construction and program debugging are *integrated*, providing a uniform user interface. That interface is *language-oriented* in that all communication takes place in terms of language constructs, and only the source representation of the program is visible. LOIPE is *display-oriented*, i.e., information is displayed in an organized manner. LOIPE is *compiler-based*, i.e., it translates programs into machine code and executes them in a manner that permits support of program development for different target machines. Nonetheless, LOIPE provides flexibility and fast response, informing the user of inconsistencies while he is still in context. In the following sections we argue for the choice of each of these characteristics.

1.2. Integrated Programming Environments Vs. Toolkits

Programming facilities can be provided in different ways. One possibility is to implement each facility as a separate tool and present the programmer with a *toolkit*. This is illustrated in Fig. 1-1.

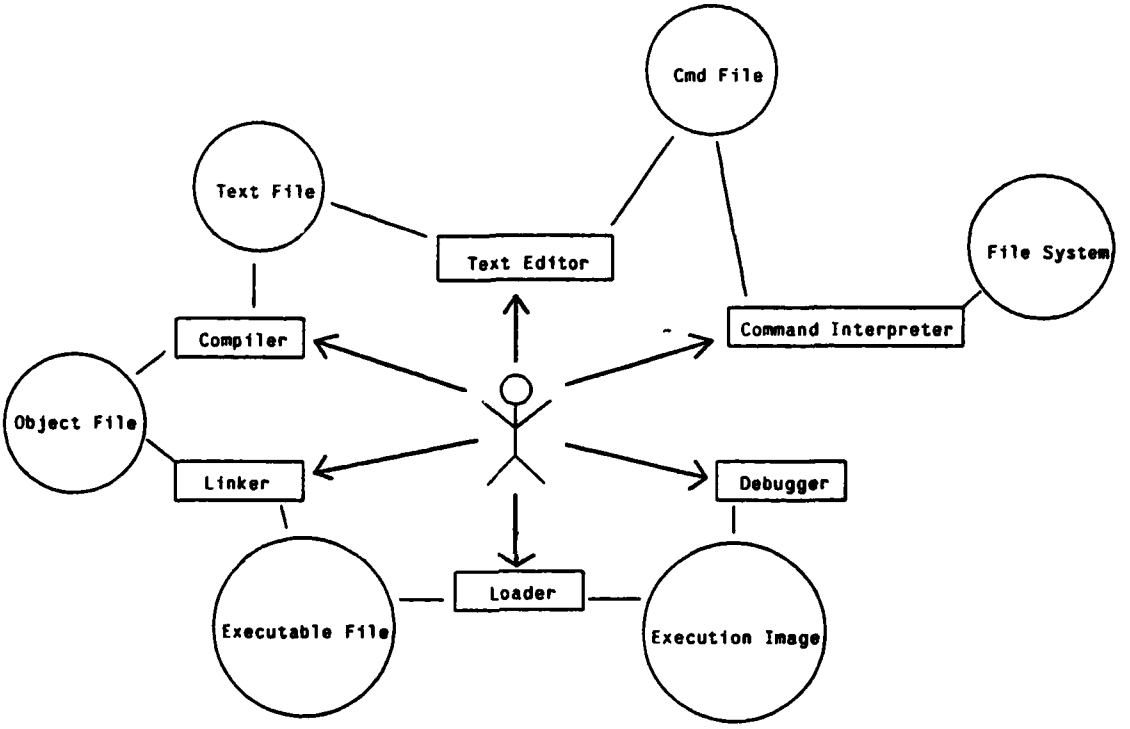


Figure 1-1: Traditional Compiler Environment

An example of this approach is the Programmers Work Bench [Ivie 77]. In the toolkit approach the user invokes the tools explicitly and applies them to various representations of the user program. The toolkit does not restrict the use of the tools. The programmer can apply tools at arbitrary times or not at all, generating inconsistent versions of the program. There are rules that guarantee consistency of the program representations after program pieces have been added or modified. Some toolkit systems provide tools that allow certain rules to be enforced, e.g., the Unix *make* facility [Unix 81a]. These tools are also applied at the user's disgretion.

In a toolkit tools are implemented as separate and independent programs. They communicate only through textual or intermediate representations. First, this results in

considerable duplication. For example, both the compiler and the pretty-printer have to "understand" the syntax of the language, thus contain a parser for the same language. Second, different tools often require the user to be aware of different views of the program. A text editor refers to lines and pages, a link-editor to external references, pages and segments. Third, tools often serve several purposes and, therefore, can provide only the least common denominator. For example, a text editor, used for both document preparation and writing of programs in different languages, interacts with the programmer in terms of a text file - independent of the actual contents.

The second approach, which has been chosen for LOIPE, is an *integrated system* approach. Tools are integrated into one system and are tuned to serve a common goal, share information, and use a common terminology. The user only sees the source program representation and interacts with a single user interface during the whole programming process. In the case of LOIPE this interface is a structure editor. Other system tools are automatically called by the interface and are hidden from the user. Similarly, the user is not aware of program representations other than the source program, which in LOIPE is the program text generated from the program tree. This is illustrated in Fig. 1-2.

An integrated programming environment overcomes problems of the toolkit approach by using knowledge of the environment and the tasks to be performed. The programming environment takes responsibility for maintaining all program representations, i.e., the *program data base*, in a consistent state. The program data base is considered to be in a *consistent state*, if all effects of a user modification on other source program pieces and derived information, such as semantic information or object code, have been propagated. The state of consistency includes state information as to whether a program part is incomplete or contians semantic errors. The program data base is kept in a consistent state by controlled application of tools. The system knows what tools exist and what their effect on the program is. The use of a tool on the source representation of a program piece may invalidate a derived representation, e.g., the object code. It must be regenerated before the program can be executed.

Integration of tools allows them to be tailored to specific tasks. For example, the program editor may have knowledge of the language syntax, allowing the user to build a program out of language constructs rather than lines of characters. For display purposes, the editor may have some knowledge about the formatting conventions in this language. The editor may

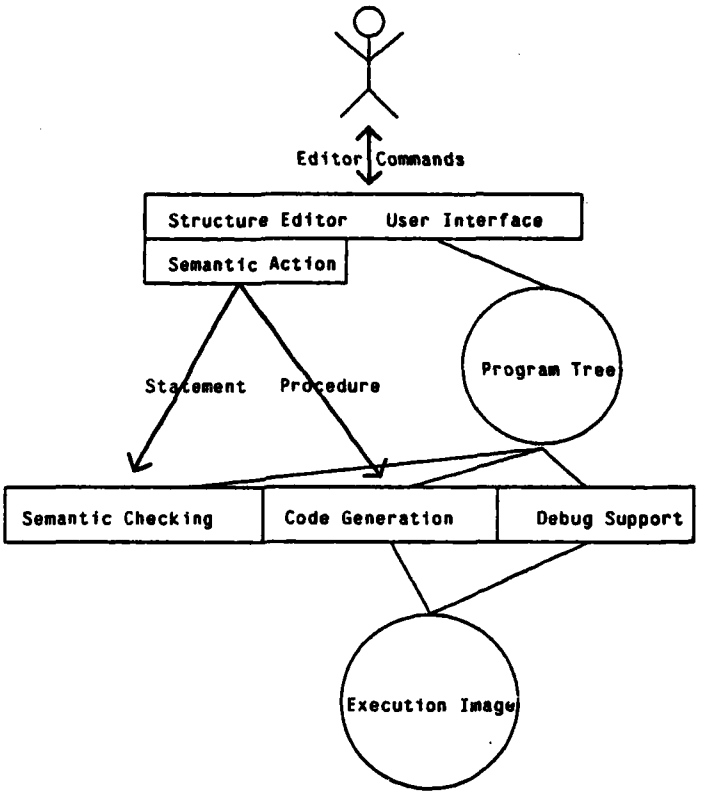


Figure 1-2: The LOIPE Environment

explicitly share information with the compiler, i.e., that the editor produces syntactically correct programs, thus simplifying its implementation. The display and program construction mechanisms of the editor can be used as a front-end through which other tools communicate with the user. A debugger, for example, can display the program execution state and allow setting of conditional breaks in terms of the programming language. Thus, in an integrated system, language knowledge, program information and processing mechanisms can be shared between the tools, avoiding duplication. The result is a uniform view of the program and its execution in terms of the source program, and uniform communication between the system and the user using the vocabulary of the supported language.

1.3. Language-Oriented Vs. Generic Systems

The programming language provides the means to describe a program. This should be the only program representation through which the user and the system communicate. In a language-oriented system the source program as well as the execution state is examined and manipulated at the level of the programming language, i.e., at the abstraction levels introduced by the user in the program. To do so, all system facilities that interact with the user must have embedded in them information about the syntax and semantics of the programming language. That information can be included to various degrees and in different forms.

- A tool may be totally *language-independent*, i.e., has no knowledge of the language on hand. Such tools can support different languages. One example is the text editor, which makes no assumption about the structure of the textual information. Thus it cannot aid the programmer in writing programs that adhere to the language specifications.

- A tool may be *language-related* in that it makes certain assumptions about a programming language, that are satisfied by a class of languages. The Language-independent Symbolic Debugging System [Johnson 77] and the VAX/VMS debugger [VMS 78] are able to handle multiple source languages. They understand language structures, such as procedures, statements, objects, that are found in many languages. However, the set of languages is limited in that their runtime support must be compatible. The user interacts with such a tool in terms of concepts that exist in several languages, but are not specific to any language.

- A tool may be *language-specific* or *language-oriented* in that it has full knowledge of the language structure. A prime example is the parser of a compiler. The information about the language can be encoded directly in the tool or be provided in a descriptive form. In the first case, the tool itself is affected by a language change, whereas in the second case only the description must be updated. The description can be interpreted, such as in table-driven editors and constructors [Feiler 81a, Donzeau-Gouge 80], or used as input to a generator of the tool, e.g., a parser generator [Johnson 75].

Much of LOIPE's language knowledge is encoded in a descriptive form. Therefore, we claim that LOIPE is able to support different languages. In its realization, LOIPE supports one, but not one specific language. Chapter 6 contains a discussion of the dependence of LOIPE on language specific knowledge.

1.4. Compiler-Based Vs. Interpreter-Based Programming Environments

Programming systems fall into two major categories. Some systems perform as much checking and binding as possible statically in order to limit the amount of runtime checking. Because of the amount of processing before runtime, these systems are assumed to work in *batch-oriented* fashion. Such systems are perceived to be quite *inflexible*, especially in connection with strongly typed languages. Other systems perform most or all of the necessary checking and binding at runtime. As a result, little processing is required when the source program is modified. The system is more *responsive*. Such systems are also more *flexible* because semantic rules are not enforced until the program is executed.

Compilation tends to be associated with the first kind of systems, whereas *interpretation* is viewed as the implementation technique for systems of the second kind. Furthermore, it is often assumed, that systems with static processing, i.e., compiler-based systems, result in efficient program execution due to the transformation of programs into a representation that is directly interpreted by the hardware. Interpretive systems are less efficient at runtime due to software interpretation. Thus, interpretive systems have been viewed as a tool for experimental and prototype programming, whereas production software is developed with compiler-based systems, which often provide relatively poor programming support.

It seems to be an intrinsic property of every major software that it will be modified as long as it is used. Program correction, adaption to specific environments, and improvements contribute to this "Law of Continuing Change" [Belady 78]. Because of the continuing change, it is desirable to combine the flexibility of the interpretive system and the efficiency of the compiler system into one programming environment.

We believe that it is not inherent to a compiler-based system to be inflexible and noninteractive, but it is a property of a particular implementation. In this dissertation we show that with appropriate mechanisms a purely compiler-based system can perform as much static checking and binding as possible, yet maintain flexibility and fast response. The behavior of such a system appears to the user to be similar to that of an interpretive system.

The compiler-based approach has an advantage over an interpretive system. Compiler-based systems allow the host/target approach of program development (incidentally one of

the requirements for a programming environment in Pebbleman [DoD 78]). The executable representation can be executed independently from the source program, i.e., on a different machine than the one on which the source program resides. The object code is generated on the host machine by a cross-compiler, see for example Bliss11 [Wulf 75]. Minimal support resides on the target machine for loading and executing the program. An interpretive system, in contrast, requires the interpreter and the source program in internal representation to be resident on the executing machine. This means that the interpretive system must be ported to the target machine. The interpretive system itself has certain requirements on the hardware, e.g., the availability of a disk, that must be satisfied.

1.5. Display-Oriented Systems

CRT and raster scanned display terminals provide the facility for random manipulation of the two-dimensional display screen. This facility, however, was rarely taken advantage of in programming environments (with the exception of screen editors). An early example of a screen-oriented debugger is RAID [Petit 69] which allows display and monitoring of variables at fixed screen positions. In COPILOT, an interactive programming system, Swinehart later used multiple CRT displays to provide the user with several contexts simultaneously [Swinehart 74]. With the appearance of personal computers with high bandwidth display, e.g. Alto [Thacker 79], the use of two-dimensional display has been exploited to a larger extent. Both the DLISP system [Teitelman 77] and the Smalltalk system [Ingalls 78] use the notion of windows, an idea initially proposed by Kay in 1969 [Kay 69].

Windows are regions that may be overlapped on the screen. Each window provides a display area that is maintained independently of other windows. These windows are used to organize information. They provide a facility to to maintain program editing, program execution, and debugging contexts simultaneously. Special windows, called *menus*, list a set of commands which can be invoked with the cursor.

A pointing device permits the user to refer to arbitrary locations on the screen. This pointing device is being used to perform window selection, cursor repositioning, and command invocation by menu selection. Although, we consider a pointing device desirable for user interaction, we do not include it as a necessary communication medium for LOIPE. We believe that even in a system with such a pointing device the user should be able to achieve the same effect by using only the keyboard.

The high bandwidth of the two-dimensional display has been further explored to provide alternative display forms in addition to textual display. Graphics facility permits pictorical display of data structures and analog display of the execution [Myers 80, Model 79, Yarwood 77]. In this dissertation we are not concerned with the mechanisms for providing two-dimensional display and a window management facility for different types of display hardware, but assume that they are available. However, certain properties of such a window management facility that are desirable for LOIPE are discussed in section 2.2.

1.6. Review of Previous Work on Programming Environments

In this dissertation the review of programming environments is limited to programming support for individual programmers. Many articles have been written on this subject, including some that provide a good historical survey of programming and program debugging, e.g., [Gaines 71, Blair 71, Satterthwaite 75, Model 79]. A large number of contributions, however, are limited to a single tool in a programming environment, for example to program editing, e.g., [Donzeau-Gouge 80], user friendly compilers, e.g., IBM's checkout compiler [Warren 75], or programming languages with support for construction of larger software systems [Wirth 77, Lampson 77, DoD 80]. In this review section we restrict ourselves to pointing out contributions to the overall concept of an integrated programming environment, in which tools are built to cooperate in supporting the programmer. References to existing work on individual parts can be found throughout the dissertation.

In the next two sections we examine two programming environments in actual use. One is a traditional compiler-based programming system, and the other a sophisticated interpreter-based system. The specific systems have been chosen as an illustration of the facilities that can be expected in programming systems. In the third section individual contributions to the advancement of interactive programming environments in the style of LOIPE are reviewed.

1.6.1. Traditional Compiler-based Programming Systems

One of the more advanced programming systems in day-to-day use that is based on a compiler is the Mesa system [Mitchell 79]. It consists of a high level programming language with strong type checking, data abstraction facilities, and the concept of modules, a text editor, a compiler with separate compilation, and a display-oriented, interactive source

program debugger. Much of the leverage in this system is gained through the power of the language, its enforcement by the compiler and the debugger's ability to show the user both the source program, and the execution state in terms of the user defined symbols and data types.

In addition to the high-level source language, the Mesa system supports a configuration language, C-Mesa [Mitchell 79]. This configuration language permits the user to express the interconnection and dependency of different software modules that comprise a runnable system. Using this description the Mesa system determines all modules that have to be processed in order to restore consistency of the program representations. The processing, i.e., compilation and link/loading, is performed in batch form. In LOIPE the dependency of software modules is recorded in the program tree as part of the semantic information. Consistency of the program data base is restored incrementally.

In the Mesa system, several of the tools cooperate in that they know of each others existence and make use of their knowledge. The compiler provides extensive information about the program for both the configuration processor and the source program debugger. The debugger works at the level of the source program. All references to program locations are expressed in terms of the source program text, the text line being the unit of reference. Debug commands are invoked relative to the source code lines. The debugger is able to evaluate source language expressions that are entered by the user, and show the program state in terms of the data abstractions defined in the program. The debugging facility of Mesa takes advantage of the ability to organize displayed information in different windows or contexts on a display screen, and the ability to refer to any location on the screen through a pointing device. The debugging support does not affect the speed or size of the executing program, if the debugger is not invoked. The Mesa debugger, however, lacks some functionality that can be found in other debugging systems, such as single stepping and conditional breakpoints [Unix 81b], or monitoring of variables and abortion of procedure invocation [Lane 73]. In contrast to existing debuggers the LOIPE debugging facility is language-oriented. Its functions are expressed in terms of the abstractions provided in the source program, and their expressive power grows with that of the supported language.

The Mesa programming system has taken the toolset approach. Thus, it is not fully integrated. Even though an effort has been made for some tools to cooperate, other tools are quite independent of the programming system. A prime example in the Mesa system is the text editor, which assumes no knowledge of the information being manipulated.

In contrast to the LOIPE system, the Mesa system is not fully interactive, even though some of its tools provide fast response. Program editing and program debugging are interactive. Switching from program editing to program execution, however, requires larger amounts of batch-type processing.

1.6.2. Interpretive Programming Systems

Due to their nature, interpretive systems are interactive and integrated. Any program part can be evaluated at any time, and all support facilities are provided through one system. One of these interpretive systems, called BASIC, has become familiar to many users through the mass marketing of home computers. The interpretive system with the most extensive programming support is Interlisp [Teitelman 78]. It is heavily used in the Artificial Intelligence community for experimental software, as are the many other Lisp systems, e.g., UCILisp [Perdue 74], and FranzLisp [Unix 81a].

As an interpretive system, it permits quick alternation between program construction and program debugging. Program editing can be performed in terms of Lisp structures. Emphasis is on sophisticated debugging and monitoring facilities, including the ability to undo certain actions. However, no guarantees are made for the continuation of execution after user modifications to the source program.

Over the years packages have been added to enhance both the programming language and the support system. One example of language enhancement is some additional support for data abstraction and modularization, commonly found in modern high-level languages. Similarly, support packages such as DWIM and Masterscope improve the interaction between the user and the system. A special version of InterLisp, called DLISP [Teitelman 77], provides a display-oriented system that takes advantage of the display and pointing device of a personal computer in a manner similar to the Mesa system.

Interpretive systems tend to have slower program execution than compiler-based systems due to program processing at runtime. InterLisp attempts to overcome this handicap by providing a hybrid system, i.e., an interpretive system in which program parts can be compiled in order to improve efficiency. This approach, however, has some disadvantages. The runtime system must be able to accommodate the execution of both program representations, an increase in its complexity. A certain amount of interpretation is necessary, even if all

program parts are compiled. Another problem concerns the equivalence of program behavior under interpretation and compilation, and the transition between the two executable representations. The Interlisp system [Teitelman 78] requires the user to explicitly specify which program parts should be compiled.

By comparison, LOIPE system attempts to provide an environment with characteristics and functionality that is very similar to that of the InterLisp system, i.e., an integrated, interactive, and flexible system. LOIPE differs from InterLisp in that it supports modern programming languages with abstract data types and modularization. Furthermore, the implementation of LOIPE is solely based on compilation technology.

Recently, several interactive programming systems have been developed for high level languages that are normally implemented in traditional compiler-based systems. All those systems, however, perform a certain amount of interpretation. The most notable system is the Cornell Program Synthesiser [Teitelbaum 80], which supports PL/CS, a small subset of PL/1. This system has some similarities with LOIPE in that it provides a language-oriented, syntax-directed program editor with integrated facilities for tracing and debugging. The program is maintained in an internal representation, which is interpreted at runtime. The Synthesizer is intended and used for small programs, such as written by students in introductory computing courses. A similar system has been built for Pascal [Shapiro 80]. Finally, the COPE system [Archer 81] is another interpretive system for PL/CS, which in contrast to the Cornell Program Synthesizer uses incremental parsing techniques rather than a syntax-directed editor.

1.6.3. Individual Contributions to Programming Environments

1.6.3.1. Mitchell's Thesis Work

Over the years, a whole series of studies has been done to investigate the improvement of programming support and communication between the user and the machine. One of the earlier works is a dissertation by J.G. Mitchell, entitled "The Design and Construction of Flexible and Efficient Interactive Programming Systems" [Mitchell 70]. Mitchell was concerned with the long turn-around time for program modifications in compiler-based systems. He proposed a system that has the flexibility of an interpreter and the efficiency of a compiler. In this system an internal representation of the program is interpreted. However, as

program pieces are executed for the first time, code is generated as a side effect of the interpretation. These code pieces are executed directly when control flow passes through the program piece again. Thus, the executable program representation converges to a representation with all program pieces in compiled form. These pieces, however, are dynamically linked together through interpretation. This idea was later applied by Hansen [Hansen 74] to improve the code quality of frequently executed program pieces by applying various optimizations automatically.

Flexibility is achieved in Mitchell's system through incremental parsing and code generation. After a modification of the program text, the extent of the modification is determined by the system in order to find all affected program parts and to reprocess them incrementally. The thesis contains an extensive study of several languages at the time (1970), and their implementation in such a system. The author also suggests certain language design considerations to simplify the implementation in flexible and interactive systems. Since the publication of the Mitchell dissertation, languages have evolved, which support the concept of localization of information and modularization [Parnas 72]. By making use of the module interface information and the interconnection structure of modules, incremental checking of even large programs becomes a possibility [Tichy 80].

1.6.3.2. Swinehart's COPILOT System

The COPILOT system [Swinehart 74] is another attempt to provide a flexible and interactive system, but use compilation to generate an executable representation. The source program text is the only visible program representation. The system, however, maintains several other internal representations and mappings between them, including executable machine code. The mechanisms for detecting modifications and determining all program parts to be reprocessed is based on Mitchell's work. In the COPILOT system, processing of these parts is not delayed until execution time. Each statement in the program is compiled individually and the generated code is placed in separate code segments. These code segments are mapped directly into segments of the underlying operating system, relying on it to perform the necessary dynamic binding at runtime.

The emphasis of this dissertation, as compared to COPILOT, is more on user interface issues and some characteristics of the debugging support. Several CRT screens are used for display of information to the user. The display space is subdivided into different windows. One window contains the user program output, a second window provides access to the

program text, and additional windows are used to orgranize the display of execution state information.

Swinehart argues that the debugging system should always be in full control over the executing user program. Therefore, the two reside in different processes. The debugging facility provides break and trace points, examination and monitoring of variables. These functions are implemented in a manner similar to the advise facility in Interlisp. Between each statement there is a placeholder, through which debug statements are threaded into the executable program. These placeholders are generated by the system independent of the application of debug statements, at a considerable cost in program size. The COPILOT system is interactive, because the user can change beteen program modification and program debugging without considerable delay. However, the system does not support continuation of execution after user modifications.

The COPILOT system differs from LOIPE in several ways. COPILOT provides a source program view, yet is not language-oriented, i.e., it does not communicate with the user in terms of language-constructs. COPILOT relies heavily on the underlying operating system to support the incremental update of the executable representation. With the choice of a statement being the unit of replacement puts high demands on the segmentation system in that every statement resides in a separate segment. The executable image of the LOIPE system does not place such demands on the supporting operating system and hardware. The COPILOT debugging facility works at the source text level, but not in terms of the constructs in the language. Debug statements are added through special commands, increasing the complexity of the user interface. It lacks some of the functionality that is provided in the LOIPE debugging system, most prominently the ability to resume execution after user modifications.

1.6.3.3. Model's Monitoring System

Model's thesis [Model 79] discusses program debugging in the context of Artificial Intelligence applications. He introduces the term meta monitoring, which refers to the ability of the system to communicate with the user in terms of abstractions expressed by the user in the program. A source-level debugger is proposed that accepts complex queries such as "whether anything unusual has happened during execution" [Model 79].

The system interprets debug events that are generated by a program run, and presents the user with information that was requested. This may require recognition of differences

between discrete program states and active processes. The information may be organized into different windows, highlighted by using different fonts, or shown in graphical form. This approach to debugging is based on the assumption that for each user system, there is a small set of fundamental structures and operations on them. It is claimed that a fairly complete, high-level description of the system activity can be generated by associating events with these structures and operations. The stream of events then becomes the input for the debugging monitor. A prototype system, built on top of DLISP and taking advantage of many of DLISP's facilities, demonstrates some of the ideas by providing monitoring support for two large AI systems.

LOIPE is able to support such an approach to debugging in that it has a language-oriented debugging facility that grows with the power of the supported language. Through dynamic assertion checking and the ability to associate assertions with procedures, objects, or even data types, related high-level program state information can be monitored (see chapter 3).

1.6.3.4. Deutsch's Interactive Program Verifier

The *Interactive Program Verifier* of Deutsch's dissertation [Deutsch 73] has many traits of a programming environment, even though its goal is to support interactive program verification. The editor has some knowledge of the language structure, and attempts to correct mistyped or misspelled keywords or identifiers. A canonical internal program representation is used for both efficient storage and efficient manipulation. Based on this internal representation the proof control mechanism interactively guides the user through proof steps, permits entering and modification of assertions and allows changes to variables. This work indicates that methods for program verification can be carried over to enhance language-oriented program debugging by supporting interactive dynamic assertion checking with appropriate expressive power for the assertions.

1.6.3.5. The Cedar Project

The Cedar project [Deutsch 80] is an effort to design and implement an advanced program development system that satisfies the needs of researchers currently using Mesa, InterLisp, and Smalltalk. This project includes a programming environment as one part, in addition to support for managing multiple versions and configurations and for project management. Major concerns in the first phase of the project are the design and implementation of a language that encompasses the capabilities of Mesa and InterLisp, a user interface supports

system that is well-separated from the application program, and an entity-based data base that manages all information about the program and coordinates modifications activities [Cattell 79]. For the programming environment support initially a traditional approach based largely on the Mesa system is taken. In a later phase more sophisticated programming support with an integrated program manipulation facility is planned to be included [Deutsch 80].

1.7. Plan of the Thesis

LOIPE takes a novel approach to compiler-based programming environments. This approach can be discussed along three dimensions:

- The user's view of LOIPE can be treated separately from the technical issues related to the implementation of LOIPE solely by compilation.
- LOIPE's support for incremental program construction, i.e., editing and compilation, can be separated from its language-oriented debugging support.
- Both the design of LOIPE and an evaluation of LOIPE based on a prototype implementation can be discussed.

LOIPE from the designer's point of view is discussed in the next four chapters (chapters 2 - 5). Chapter 6 elaborates on the prototype and evaluates the feasibility of LOIPE's approach. The design discussions of LOIPE are divided into two parts: issues relating to the user's view in chapters 2 and 3, and problems of implementing an interactive programming environment through compilation in chapters 4 and 5. Incremental program construction is addressed in the first chapter of the two design parts, i.e., chapters 2 and 4, whereas chapter 3 and 5 concentrate on integrated, language-oriented debugging in the programming environment.

In chapter 2 the user's view of program construction is presented. First, a crucial element of LOIPE, the structure editor ALOE is introduced. For a full discussion of ALOE itself we refer to [Medina-Mora 82]. Here we point out those mechanisms of the structure editor that are important to the design of LOIPE. The chapter also shows how these mechanisms are used to provide the user with an incremental program construction facility that is flexible and takes up the responsibility for certain chores such as storage of the program in permanent storage and consistent update of the program data base.

Chapter 3 elaborates on the user's view of a program debugging facility that is integrated

into the program construction facility presented in chapter 2. By doing so a new dimension is added to debugging. As in some interpretive systems the distinction between debugging and program manipulation diminishes. Furthermore, language-oriented debugging support can be provided, whose power grows with the power of the supported language.

Chapter 4 discusses the modification cycle *edit, compile, link, load* in the context of LOIPE. The steps of the modification cycle are performed between user interactions in order to restore consistency of the program data base. Fast response is guaranteed through incremental semantic checking and partial replacement of program pieces in the executable representation.

Chapter 5 addresses problems of supporting language-oriented debugging in the context of incremental program construction. Issues such as continuation of execution after user modifications and support of optimizing code generators are discussed.

Chapter 6 evaluates the LOIPE design and a prototype implementation. In the process of discussing the support of the programming language Ada the localization of dependence on a specific language is pinpointed and LOIPE's potential as a system for generating interactive language environments is illustrated. The feasibility of the LOIPE approach is shown through some measurements on the LOIPE prototype.

Chapter 7 concludes the dissertation with a summary of the contributions and a list of topics that will require further investigation.

Chapter 2

A User's View of Program Construction

The user manipulates the program in terms of the programming language. He is concerned only with the source code representation. It is modified through an editing system with language-specific knowledge. The program appears to be structured by the constructs in the programming language, such as procedures and modules.

The actual storage of the program in files is of no concern to the user. It is irrelevant whether every procedure or every module is kept in a separate file. Similarly, it is of no concern to the user when and how changes to the program are reflected in the file system as long as the system can provide a consistent view of the changes. Furthermore, some LOIPE components, such as the semantic checker, code generator, and linker/loader, are invoked automatically. The user does not have to remember when to invoke a tool.

The user enters and modifies the program through a syntax-directed editor. The user fills in language constructs for which the editor supplies the concrete syntax. The editor enforces the syntax of the language by permitting only language constructs that are legal at any time. As the program is being entered or modified, the system constantly checks its semantic correctness. The user is informed of errors, but is not required to correct them. Similarly, the user is permitted to leave any piece of the program incomplete although syntactically correct. The system also takes an active role in informing the user of the potential effects of a modification on other program parts, and allows him to reconsider the change. If the user performs a modification, all affected program parts are rechecked for semantic consistency. Once an error is detected, the system may propose a correction which can be accepted by the user through confirmation. An example is the filling in of the appropriate declaration for undeclared variables.

Program parts are transformed into an executable form as they become semantically correct. At any point in time the user can attempt to execute even incomplete programs without delay. Execution will proceed until an incorrect or incomplete program part is reached. It is the system's responsibility to keep the executable program consistent with the source code representation. The discussion of monitoring and debugging facilities is deferred until chapter 3.

After this bird's eye view of the system we continue with section 2.1 by discussing our choice of a structure editor as the basis of the LOIPE system and by pointing out the properties of the chosen structure editor ALOE that the LOIPE design depends on. In the remaining three sections we elaborate on LOIPE's ability to provide a display-oriented environment, to provide the flexibility of an interactive environment, and to be an active participant in the programming task.

2.1. LOIPE's Language-Oriented Program Manipulator

In a language-oriented system the user views the program in terms of the supported programming language. The program manipulation facility has some knowledge of the programming language and can assist the user in the programming task. There are basically two approaches to language-oriented program manipulation. They are referred to in this dissertation as the *text approach* and the *structure approach*. In both cases the user sees program text. The difference in the two approaches lies in the way the user manipulates the program.

2.1.1. The Text Approach

In the text approach any character can be referred to and modified. The rules for moving through program text follow the rules for character strings, namely movement to the previous or next character, word, or line. The cursor refers to a single character position. Some text editors are equipped with a certain amount of information about the program structure. This information is used to perform some pretty-printing and to allow insertion of text templates for language constructs [Gosling 81a]. For program modification, however, little more than free-form text editing is provided.

The syntactic knowledge of the supported language is embedded in the program parser. It

processes the program text in order to recognize the syntactic structure of the program and to check for syntactic errors. The result is an intermediate representation that is used by other processing units, such as semantic analyzer, code generator, or pretty-printer.

Incremental parsers have been investigated in order to limit the processing cost of the parser to the extent of the program modifications [Mitchell 70, Ghezzi 79]. The incremental parser determines how much can be saved from the previous analysis when provided with the location of the modification from the text editor in an attempt to minimize reprocessing. Some parsers have been extended to inform the user interactively of syntax errors [Day 79, Wilcox 76], or to provide guidance for the correct syntax [Pinc 73].

2.1.2. The Structure Approach

In the structural approach the user perceives the program as tree-structured text. The structure is manipulated in terms of syntactic units. The user moves through the program according to syntactic units, for example from an *if* statement to its enclosing syntactic unit, to one of its components (e.g., the condition of the if statement), or to the preceding or succeeding statement. The current cursor position is indicated by high-lighting the whole construct (see Fig. 2-1).

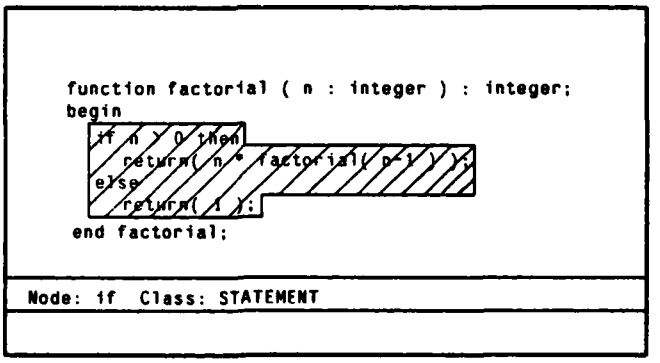


Figure 2-1: Area Cursor Display

In this approach the editor has full knowledge of the language syntax. It can enforce the construction of syntactically legal programs, and provide guidance during the construction. At any point only a certain number of syntactic constructs can be applied legally. Upon an

application of such a construct (by typing its keyword) the editor fills in all syntactic information that is not program specific and then returns control to the user to complete the missing parts. The syntactic information that is filled in by the editor can only be referred to in one unit. Individual pieces, e.g., a keyword or terminator, cannot be manipulated by changing characters, but only by conversion into different syntactic constructs. Thus, even though the user perceives it as structured manipulation of text, he actually is dealing with a *parse tree* [Lasker 74] or an *abstract syntax tree* [Donzeau-Gouge 80, Medina-Mora 82]. This tree structure is the primary program representation, and the textual representation is generated by the program manipulation facility in a pretty-printed form. This is the only textual representation visible to the user.

2.1.3. Text Vs. Structure

For LOIPE we chose the structure approach over the text approach for several reasons. First, in the structure approach the syntax-directed editor or *structure editor*, has full knowledge of the language syntax. The user communicates with the system in terms of language constructs rather than in terms of characters and lines. Since the structure editor "knows" the syntax of the language it relieves the user of the burden to remember the syntactic details such as key words, separators and terminators. They are automatically provided by the editor as a construct is applied. Since these do not need to be typed, they cannot be mistyped or forgotten. As a result, the structure editor requires fewer characters to be typed and reduces the number of mistakes that a user can make. However, the structure editor requires the user to always think of the program as highly structured information. It does not permit free-form manipulation of the program text. The mental effort for manipulating the program in a rigidly structured way is different and may be higher than that for plain text manipulation. Some editing systems combine both editing approaches. For example, in the Cornell Program Synthesizer [Teitelbaum 80] the overall program structure is manipulated in a structured way, but expressions are treated in free-form. The impact of the structured style of program manipulation on the programmers is not discussed here, but is being studied in several structure editor projects, e.g. [Medina-Mora 82, Teitelbaum 80].

The second reason for our choice is that the text approach would require the system to either maintain both the text representation and an internal structured representation, e.g., when using an incremental parser, or to rederive the structured information over and over from the text representation for syntactic and semantic processing. The structure approach

requires only a single representation, a tree. This representation is used as the common program representation for all other LOIPE parts. The textual representation of the program part visible to the user is generated dynamically from the program tree by a process called *unparsing*. It performs the inverse function of a parser. The unparser maps every element of the internal structure into a text template. Such a mapping, which defines the concrete syntax of a construct, is called an *unparse scheme*. Due to the dynamic generation the program text does not have to be stored in the program data base. The recurring cost of generating the program text is limited by the small amount of program text visible on the display.

Third, the structure approach permits different textual representations to be generated from the same program tree. We refer to them as different *textual views* of the program or *program views*. A textual representation is dynamically generated from an internally structured representation. By associating different unparse schemes with the same structured representation, the unparser is able to produce different textual views of the program expressed by the internal representation. An unparse scheme affects the textual view in the following ways.

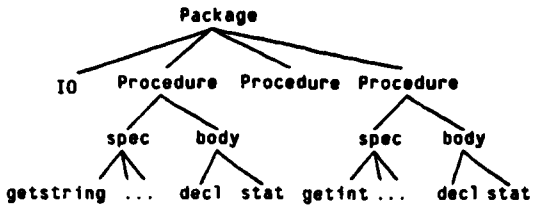
1. Hiding
   The tree structured program representation can be unparsed to various depths. Program parts below a certain depth are not shown explicitly, but indicated in form of ellipses or named labels. This permits programs to be shown at various levels of details. Different approaches and techniques for the provision of this facility can be found in [Teitelbaum 80, Donzeau-Gouge 80, Mikelsons 81].

2. Pretty Printing
   The unparse schemes contain formatting information. This information is interpreted by the unparser when generating the textual representation. The result is a formatted program text. Thus, the unparser has the effect of a pretty printer.

3. Concrete Syntax
   By mapping elements of the internal representation into text templates, the structure editor gives the user the impression of filling in forms. Since the text templates are defined by the unparse scheme, different concrete syntax can be specified for the same abstract tree.

4. Selective Views
   The unparser can limit the view of a program by selecting only a subset of all offsprings to be shown in the program text. For example, the text form of both the specification and the implementation of a module (package) can be generated from the same program tree. When the unparser displays in specification display mode the implementation of the module is not accessible. Fig. 2-2 illustrates the use of limited views for an Ada package text representation. Furthermore,

offsprings can be shown read only, i.e., cannot be reached by the cursor for modification.



```
Package IO is

   procedure getstring(str:string);
   ...
   procedure getint(val:integer);

end package IO;
```

```
Package Body IO is

   procedure getstring(str:string);
   begin
     ...
   end getstring;
   ...
   procedure getint(val:integer);
   begin
     ...
   end getint;
end package IO;
```

Figure 2-2: Two Textual Views of a Package Tree

This is not to say that in the text approach the generation of different textual views is not possible. However, the cost for doing so is much higher, since original program representation is an unstructured text representation.

Finally, the structure editor has an advantage over the text editor in that for every editing operation context information is available in an internal structured form. This permits the system to perform context sensitive processing in small steps to respond to user actions intelligently, and to actively contribute to the programming process, as the user is manipulating the program. The use of an incremental parser in the text editing approach would also allow the system to pinpoint the modifications in the program tree representation. However, the program text must be submitted to the parser frequently - possibly after every keystroke since the text editor only knows strings of characters - in order for the system to respond while the user is still in context. Furthermore, mechanisms must be provided to partition the program text into smaller units than the whole program in order to limit the amount of text to be scanned by the incremental parser, especially for large programs.

2.1.4. The ALOE Program Manipulator

ALOE, a syntax-directed editor, which was designed and implemented by Raul Medina-Mora [Medina-Mora 82], has been chosen over other structure editors, because it combines all of the features of a structure editor that were essential for its use as a component of LOIPE.

ALOE is *language-independent* in the following sense. An editor for a particular language is generated from a grammatical description of the language. In the grammar the structure of the abstract syntax is separated from the concrete syntax. For each production in the abstract syntax several unparse schemes specifying concrete syntax can be specified [Medina-Mora 81]. *Multiple views* can be defined for the same abstract program representation.

ALOE provides an *action mechanism*, which allows other system functions to be associated with editor operations on each of the nodes in the program tree. These functions are automatically called by the editor whenever an editor operation (for example create node, delete node, move cursor) is performed on a tree node. Thus, the editor acts as driver for the whole LOIPE system.

A *reporting mechanism* in ALOE allows a sequence of messages to be displayed to the user after an action routine completes. Each message can be associated with tree node. The program text corresponding to the subtree, e.g. a variable, statement or procedure, is highlighted by the cursor and the message is shown on the terminal screen (Fig. 2-3).

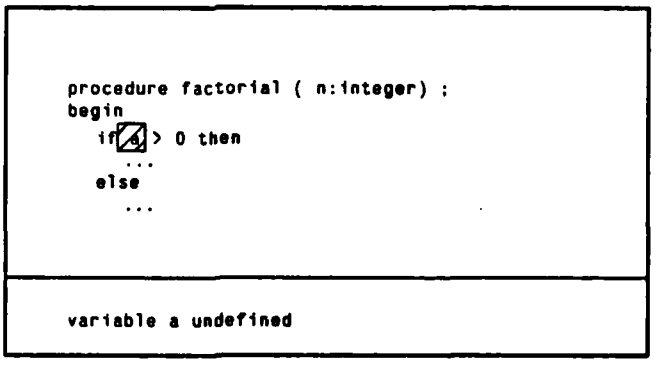


Figure 2-3: Error Reporting Through Structure Editor

This reporting mechanism provides a standard way for other system parts to communicate with the user through ALOE.

ALOE manipulates abstract syntax trees. These are more compact than parse trees, but still contain all required information. The abstract syntax tree can be obtained by collapsing and normalizing a parse tree [Donzeau-Gouge 80, Medina-Mora 82]. The structure of the abstract syntax tree used by ALOE is very similar to that of program trees defined by DIANA [Diana 81] and the external DIANA representation can be generated without difficulty through an unparse scheme. DIANA is a description of the intermediate representation of Ada programs and is used by various Ada compilers. For example the PQCC project uses it as the interface between different phases of the compiler [Wulf 80]. Since our syntax trees can be unparsed into DIANA trees we should be able to interface to code generators produced by PQCC for different machines and languages.

In the next sections we discuss how we use the mechansims of ALOE to provide a language-oriented, interactive program construction facility in LOIPE, that cooperates with the user. In section 2.2 facilities for communication of information between the system and the user are discussed. Section 2.3 elaborates on facilities that provide flexibility of programming for a compiler-based high level language with data typing. The use of the action mechanism for active participation of the LOIPE system in the programming task is illustrated in section 2.4.

2.2. Display of Information

The structure editor ALOE makes use of the two-dimensional display of CRT terminals. One example is the indication of the current cursor position in the program by a highlighted area (see section 2.1.2). Another example is the subdivision of the screen into three display areas: a command window , a message window, a help window and an edit window. The user enters all commands through the *command window*. The *message window* is used by the editor to show status information and to report errors as part of the error reporting mechanism. Status information includes information about selected editor modes and additional information concerning the cursor position. Help information such as the legal set of constructive commands on a description of editor commands is shown in a *help window*. An *edit window* shows the textual representation of a program tree according to a certain unparse scheme, highlighting the position of the cursor in that window. The user can scroll the window both horizontally and vertically over the program text. The user can also change the unparse scheme. If appropriate unparse schemes are provided by the designer of the language description for ALOE, the user can look at the program at different levels of detail.

2.2.1. Display Management in LOIPE

The organization of program display, as provided by ALOE, may be satisfactory for a simple program editing facility, but maintenance of larger software programs requires additional support. The demands on the display change from program construction to program execution and debugging. While scanning through the program the user would like to see it organized according to the abstractions defined in the program. Such a browsing facility for LOIPE is discussed in the paragraph below. While editing the program the user prefers to use the full screen for display of the program piece being modified. During the execution the screen must be rendered to the user program for display. If the program is being debugged only part of the screen can be made available for the user program because the debugger wants to communicate the progress of execution to the user. Thus, it is necessary for LOIPE to maintain several several layouts of the display screen and to allow the user to switch between them. The layouts are defined in form of a description, which can be tailored to each user. Furthermore, the user is able to adjust the layout dynamically. The details of the display management functions are dependent on the specific display device. The At [Ball 80] and Canvas [Ball 81] subsystems are two examples of such a layout management facility for raster scan displays with a pointing device. We will, therefore, not elaborate on such a support facility, but continue by discussing the implementation of a browsing facility based on structure editor mechanisms and the window support.

2.2.1.1. Display During Program Construction - A Browsing Facility

It is usually left to the user to partition a large program into files and organize them such that it is possible to maintain the program by moving through these files with a text editor. In an integrated language-oriented programming environment, the management of files should be of no concern to the user. The user should be able to move about the whole program at the level of the source program. Modern programming languages e.g. Ada [DoD 80], provide structuring facilities that support even "programming in the large." This idea of having the user manipulate a program only in terms of the structure provided by the supported language has been partially tested in a tool that is used to build both the LOIPE and the Gandalf system [Feiler 79a, Denny 81].

In LOIPE the whole program is displayed through the structure editor. By using different unparse schemes, the editor is able to show the program at various levels of details. For example, only the specification of a module may be shown. LOIPE takes advantage of this

capability by associating a different program window with each level of detail to be displayed. The levels of detail follow the abstraction mechanisms of the supported language. This is illustrated in Fig. 2-4.

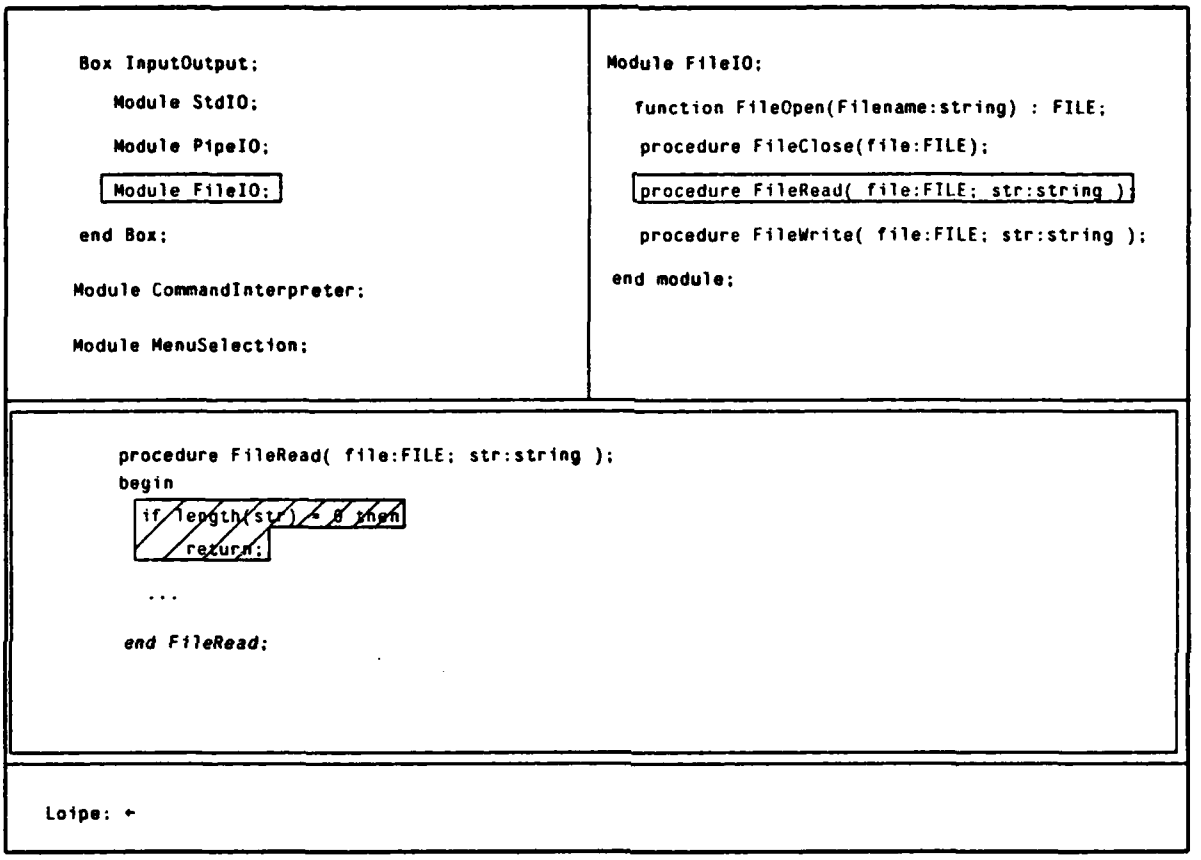


Figure 2-4: Browsing In A Modular Language

The first program window shows the top level of abstraction, namely the list of modules that comprise the program. The user can manipulate that program window as any other edit window. The cursor can be moved to one of the modules. That module can, then, be selected for display at the next level of abstraction. Selection is done by moving the cursor in at the target module. The action mechanism of the editor informs LOIPE that an attempt has been made to enter a program subtree that is currently not visible. LOIPE reacts by attaching the subtree of the selected module as the root to a new program window, the *dule window*, and displays it with a different view. The result is that the specifications of the module and a

list of its procedures, global variables, and data types appear in the module window, and that window becomes the current edit window. Similar to the selection of a module, the user can select a procedure and cause its implementation to appear in a third program window, the *procedure window*. In order to examine or modify a different module or procedure, the user just changes the current edit window to the program or the module window and repeats the selection process. The effect is that of a browsing facility in which the user moves about the program in a hierarchical fashion. Such a browsing facility exists, for example, in the Smalltalk system [Goldstein 81].

2.3. A Flexible Agent

Provided with a description of the syntax of the supported language, the structure editor enforces correct syntax by limiting the set of language constructs that can be applied at any point, and by automatically supplying the concrete syntax when a construct is applied. Since the user is not required to complete every part of the program, the resulting program representation is syntactically correct, but potentially incomplete.

This program representation must be processed further by LOIPE to determine the semantic correctness and to generate an executable equivalent. Through ALOE's action mechanism, the semantic analyzer of LOIPE is invoked incrementally. The semantic analyzer checks the modified program part for semantic errors. Once a semantic error is detected, it is recorded in the program tree and reported to the user via the error reporting mechanism. The realization of the incremental semantic checking mechanism will be discussed in chapter 4.

Semantic correctness of the program is not enforced. Enforcement of semantic correctness would prohibit certain program modifications. Because LOIPE checks for effects on the semantic correctness incrementally and records the results of the analysis for individual program pieces in the executable representation, LOIPE is able to permit execution of program that are incomplete or semantically incorrect. This gives the user the flexibility that is customary in interpretive systems. The next two paragraphs elaborate on the interaction of the semantic analyzer with the user and on the support for execution of incomplete programs.

2.3.1. Phases of Program Development

So far the semantic checker reports semantic errors as they are detected. The user, however, goes through different phases of program development. In some phases it is more hindering than helpful to be immediately notified of errors. During the construction of a program, for example, the user wants an assignment to a local variable that has not been declared. The user does not want to change the focus of attention to add a declaration. Since he is aware of the intentional inconsistency, notification should be suppressed until the user considers the construction of the procedure complete. The system can recognize that when the cursor leaves the procedure and informs the user of remaining errors.

In LOIPE the user has control over the amount of communication from the system. For this purpose semantic checking is designed to be logically independent of the error reporting. Semantic checking is performed incrementally at the grain of individual language constructs, and the resulting state is recorded as part of the program tree.

Semantic errors are reported through an *error report filter*. The semantic action mechanism invokes this filter after the semantic checker has completed. The filter decides whether to have error messages from the semantic checker displayed by the editor upon return of control, or to suppress the error messages. In the latter case the error messages are discarded, and must be regenerated by the semantic checker when their display is requested. When error reporting is suppressed no additional actions have are taken. Error messages are disposed of rather than maintained explicitly because the maintenance cost is higher than the cost of regeneration through the semantic checker.

The grain of error reporting can be controlled by the user. The filter supplies a range of different grains. The finest grain is immediate error reporting. In this case errors are reported as soon as they are detected by the semantic checker. Then there is error reporting at various levels corresponding to the different language constructs, ranging from an expression to the whole program. For example, if the grain is set to the statement level and the user leaves a statement with the cursor, the semantic action associated with that cursor movement causes semantic errors in that statement to be reported. These messages may be suppressed if no modification has been made to to the statement. Cursor movement as well as the attempt to start or resume execution of the program may cause semantic checking to be invoked and errors to be reported. In the case where the whole program is selected as the

grain of reporting remaining semantic errors are reported when execution is attempted. Finally, dynamic error reporting causes errors to be reported only for nonexecutable program units whose execution has been attempted.

In addition to suppressing error messages the error report filter must reproduce error messages when the user's focus of attention, i.e., cursor, leaves a program part of the specified error reporting grain. The semantic action associated with cursor movement leaving a node informs the filter of such a change of focus. The filter checks whether errors must be reported, i.e., whether the exited or any contained program unit is semantically incorrect. If errors must be reported the filter retrieves the error messages by asking the semantic checker to regenerate them. Then control returns to the editor whose reporting mechanism displays the messages.

Let us examine the overhead of the filter. At first glance it seems that most of the filter's work is in the check for errors to be reported. This check requires a traversal of the program subtree for which possible errors should be reported. Every node in the subtree must be visited to determine whether it is of the error reporting grain and whether it is semantically correct. The tree walk can be improved in the following way. Instead of recording the existence of semantic errors in every node, the semantic checker may record this state information only in those nodes that correspond to the various error reporting grains. These error status carrying nodes are interconnected in a linked list. As a result, the tree walk algorithm can follow this error status tree structure to find erroneous program pieces, a reduction in the number of nodes visited.

An alternative to a full search of the program tree or error status tree is a guided walk. During the error message filtering process guide markers are placed along paths leading to program pieces that are semantically incorrect. The checking process follows these paths directly to the program pieces whose semantic errors must be reported. The number of visited nodes is greatly reduced for the guided walk. The root of the subtree to be checked contains information as to whether semantic errors are contained in the subtree. Thus, subtrees without semantic errors are not traversed at all. If the subtree contains program pieces with semantic errors the walk is restricted to the part of the program or error status subtree that is marked. The number of visited nodes is substantially smaller in this restricted tree (see Fig. 2-5).

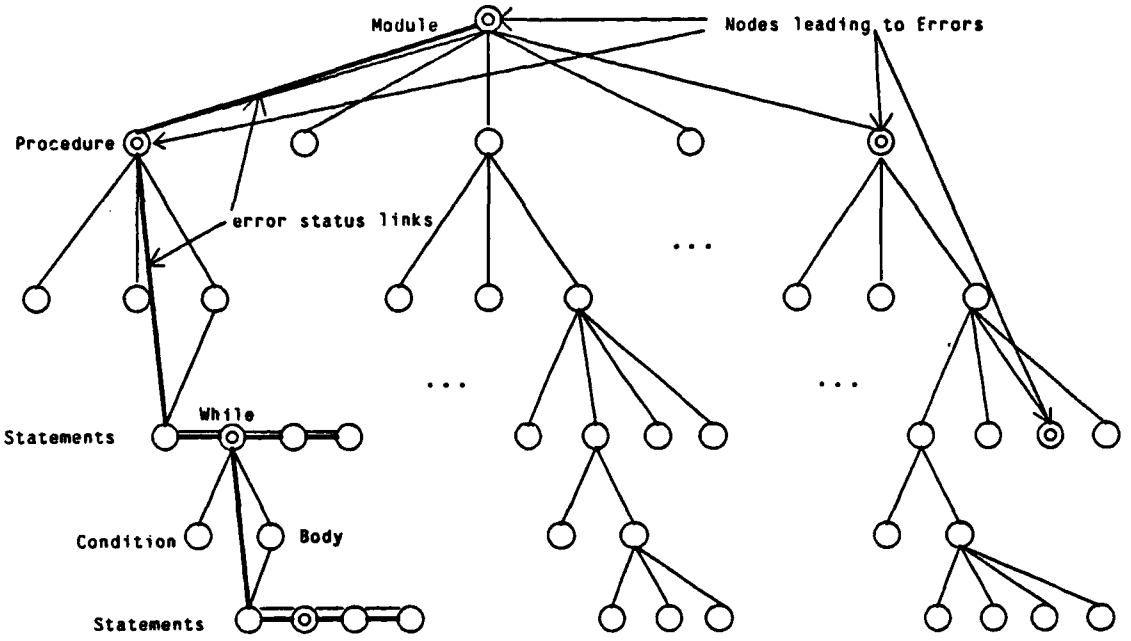


Figure 2-5: Marked Error Status Tree

During the filtering process the error status tree is marked in the following way. Each error status node maintains a count of how many of its immediate descendant error status nodes contain semantic errors. A path is marked by going to the next enclosing error status node and updating its count. The path has a maximum length of the depth of the error status tree. The root has a nonzero count if any contained error status node has errors.

A change in the error state of one program piece does not necessarily require remarking of the whole path to the root. The marking process is aborted if the count of an error status node does not change between zero and nonzero values. This is possible because the count in a error status node records how many immediate descendant error status nodes have a nonzero count rather than recording the actual number of contained errors. The denser the error status nodes with semantic errors are, the shorter is the path to be marked.

In summary, by using the marked error status scheme LOIPE is able to support different phases of program development and permits the user to adjust LOIPE's verbosity to his own needs. The cost associated with the error report filter mechanism depends only on the nesting depth of program units. It is therefore relatively independent of the overall program size.

2.3.2. Executability of User Programs

While being constructed and modified, programs may contain program parts that are incomplete or have semantic errors. In a traditional compiler-based system such programs cannot be executed. Such systems require that all program parts are compiled and linked before execution can be attempted. Interpretive systems take a different, more flexible approach to program execution. Since most of the checking is performed at runtime, the execution, i.e., invocation or evaluation of a piece of source code can be attempted at any time. Execution is suspended if an error is encountered in a program part being executed. This permits the user to freely mix construction and testing of program parts, an activity often necessary for experimental programming.

LOIPE provides a flexible system that appears to the user to behave like an interpretive system. The user can attempt execution at any time. Execution proceeds until a nonexecutable, i.e., an incomplete or semantically incorrect program part is encountered. The only difference that the user may notice is how close execution proceeds to the cause of the suspension. In interpretive systems execution proceeds right to the point in error, whereas in LOIPE execution is suspended at the entry of an enclosing program piece, which we refer to as *unit of executability*. For each such unit LOIPE determines independently whether it is executable or nonexecutable. For example, for a grain size of statement for the unit of executability a conditional statement is considered executable if the condition is complete and semantically correct, even though one of the branches may contain a nonexecutable statement. A similar interpretation applies to the grainsize of a procedure. Using this model, a traditional compiler-based system can be viewed as a system with the unit of executability being the whole program.

The procedure was chosen as the unit of executability because it is satisfactory for many practical purposes. Being an abstraction mechanism as a member of a module, the procedure defines a coherent operation on a data object. Thus, it seems sensible to stop execution when encountering such a nonexecutable operation. For executable procedures the user is able to suspend execution at smaller program units through the use of the debugging facility (see chapter 3). There is, however, nothing inherent in the LOIPE implementation that would prohibit the support of a unit of executability smaller than a procedure, e.g., a statement (see chapter 4).

2.4. Active Participation

The action mechanism of the structure editor passes control to LOIPE after every editor operation. As we have seen in the previous section, LOIPE uses this time to perform semantic checks. LOIPE can also contribute in other ways. It can guide the user through the program construction process and provide helpful information as necessary. It can also fill in some of the program parts itself, deriving them from the context. Furthermore, it can take over some of the bookkeeping chores to maintain the program data base consistently.

2.4.1. Replication of Program Parts

Modern programming languages contain quite a bit of redundancy. This redundancy is due to some of the new concepts in the languages such as module interface checking and separation of specification and implementation. For example, in the body of an Ada package the specifications of the visible part of the package must be repeated. This requires the specification to be typed or modified twice. Another example of replication is the common practice of repeating the procedure name at the end of the procedure body.

Such duplication of program parts is easily taken over by LOIPE. The more obvious way of providing such a facility is to include the replication and consistent update of all copies as part of certain semantic actions. In many cases LOIPE, however, can use the power of the unparse scheme mechanism. Procedure names can be replicated by specifying in the unparse scheme to show the program tree node containing the proedure name to be shown twice. Similarly, both the package specification and the package body of an Ada program can be generated from the same program tree, as shown in Fig. 2-2.

2.4.2. Utilization of Semantic Information

As part of the semantic analysis process semantic information is accumulated during the construction and modification of the user program. This information can be utilized by LOIPE in various ways. We will not provide a full list of possible support facilities using semantic information here, but rather point out the potential for such support in LOIPE through a few examples.

- As the user enters the procedure name for a procedure call, LOIPE uses the semantic binding of the name to retrieve the procedure specification and display it to the user to help entering the parameters correctly.

- A mistyped identifier is corrected by comparing it to the set of identifiers that is visible and satisfies the data type requirements.

- If the user attempts to make a modification to a program part, which may have side effects on other program parts, LOIPE warns the user of the extent of the possible damage done by the modification beforehand.

- Declarations for undefined identifiers can be provided at an appropriate place in the program.

2.4.3. Maintenance of the Program Data Base

So far the program representation has been described as a program tree augmented with semantic information. ALOE provides a mechanism for partitioning the program tree and for automatically storing and retrieving the partitions in the underlying file system [Medina-Mora 82]. LOIPE takes up the responsibility for maintaining the executable representation in parallel with the program tree and for storing a copy in the file system in order to expedite the generation of an executing instance of the program. This involves stepwise processing of the program tree as it is being modified to generate an executable representation, namely code generation and binding, updating the copy of the executable representation in the filing system, and incrementally maintaining a process address space that contains the user program for immediate execution. The technical details of this support mechanism are discussed in chapter 4.

2.5. Summary of the Program Construction User View

LOIPE has taken a new approach to supporting program construction in an interactive manner. Some of the components and characteristics of LOIPE can also be found in other systems. LOIPE's contribution is to provide all of them in an integrated system. The system is centered around a program tree as the primary program representation and a structure editor for the program tree manipulation. The chosen structure editor has certain mechanisms that support the program display to be organized according to different levels of abstraction, and that permit the editor to act as a single user interface.

The operations on the program tree representation trigger actions in the LOIPE system such as incremental semantic checking and stepwise generation of an executable representation. LOIPE, however, does not enforce semantic correctness and permits incorrect and incomplete

programs to be executed. Inconsistencies are reported while the user is still in context. The user even has control over the grain at which the system reports inconsistencies. This provides the user with flexibility for modification of programs normally not found in compiler-based systems.

The structurre editor's action mechanism allows LOIPE to present a data-driven programming model, in which all system activity is triggered through manipulation of the program tree. As part of the system activity LOIPE takes over the management of the program data base, thus hiding both the underlying operating system and filing system from the user.

Chapter 3

Integrated Language-Oriented Debugging

In this chapter we present the LOIPE facilities for dealing with the dynamic behavior of programs. This encompasses program testing, debugging, and monitoring. *Testing* is the process of executing a program in a manner such that different parts of the program are exercised. Its purpose is to determine that an error exists, which is done by comparing the actual results with expected results. *Debugging* is the process of localizing the cause of an error. This is achieved by examining the control flow and data flow for the test case that leads to the erroneous behavior. *Monitoring* is the process of recording and displaying the progress of execution in terms of control flow and changes in the data objects. It is used to determine whether and when the program does not behave as expected, and evaluate the performance of the program.

Because the three activities depend on each other, they are treated together under the heading program debugging in this dissertation. LOIPE's support for testing is limited to the provision of a script driver which permits repetition of test patterns . The topic of automatic generation of scripts that exercise all paths of a program is outside the scope of this thesis. However, the system structure of LOIPE is a good basis for such facilities because the program tree as the central program representation provides a rich source of information about the program structure. In the context of this dissertation the discussion of LOIPE's support for monitoring is also restricted. We refer to monitoring of program execution mostly in the context of debugging, where monitoring consists of displaying the progress of execution in terms of control flow and data state changes. However, the underlying mechanisms support performance monitoring as well as will be pointed out in the appropriate places in the next sections.

In our view a program debugging facility should have the following characteristics:

- It communicates with the user in terms of the source language.
- It permits debug functions to be issued interactively.
- It presents the program execution state in terms of the abstractions in the source program.
- It executes incomplete programs.
- It controls the executing user program.
- It is integrated with the program construction facility such that the resulting system for smooth transition between the two activities without slow system response.
- It does not overhead in space or time in the executing program for unused debugging support.

The debugging facility of LOIPE has the above characteristics. LOIPE's debugging facility is closely integrated with the program construction facility that is discussed in the previous chapter. Both the *program state* and the *debug state* are integrated into the program tree representation. The *program state* represents the current execution state of the program. This includes the call chain of active procedures and the current values of data objects. The *debug state* refers debug functions that have been defined and are enabled such as break points or assertions. The integration of program state and debug state into the program tree allows LOIPE to represent the information in a language-oriented manner. Debug functions such as dynamic assertion checking at various levels of abstraction in the user program are provided. Debug functions are entered into the source program in the same manner as the program itself is modified. LOIPE automatically reflects the application of a debug function in the executable representation using the incremental program construction mechanism. Similarly, the program state is examined and modified in a structured representation that is derived from the type definitions in the source program. A program can be executed at any time, even though it may not be completed or contain errors. Program execution stops when encountering an incomplete or erroneous program piece. The user interactively makes changes to the source program, the debug state, and the program state and can alternate between modification and program execution without delay or explicit invocation of subsystems. The user has full control over the executing program, in that it can be suspended at at any time. Once the program is suspended, the user can make modifications and continue execution even after changes to the source program. Continuation may be at the point of suspension or at a previous point in the execution history.

This concludes our bird's eye view of LOIPE's language-oriented debugging facility. We continue this chapter by first discussing the integration of the debugging facility with the program construction facility through the program tree. The integration affects both the structure of the program tree and the language that the user deals with through LOIPE. The remaining three sections elaborate on the way language-oriented debugging is provided. Section 3.2 discusses the necessary extensions to the program tree for the representation of the program state and describes the user interactions that allow its display and modification. These interactions are limited to editor commands. Section 3.3 discusses LOIPE's debug state, i.e., the representation of debug functions through *debug statements* in the program tree. These debug statements profit from the expressive power of the supported programming language. The user interaction concerning the application of debug function is limited to editor commands as well. Section 3.4 elaborates on LOIPE's ability to control the program execution. For that purpose the user interface is extended with two commands to allow starting and continuation of execution. If execution is continued after a modification LOIPE must ensure that the program state is consistent with the user program. Mechanisms for detection of damage to the program state and for its correction are discussed.

3.1. Integration of the Language-Oriented Debugger

A program debugger examines the dynamic behavior of an executing program. In compiler-based systems the executable representation of a program differs from the source representation in that the program is expressed in terms of machine instructions rather than statements of the programming language. The execution state of a program exists only in the machine representation. A program debugger must be able to present the user with information from the execution image of the program.

Early debuggers, called *octal debuggers*, converted both the program and the program state from the execution image into an octal representation. It was left to the user to determine any relationship to the source program.

The next step were *symbolic debuggers*. Symbolic debuggers have the ability to generate an assembly code representation for the static program, and to interpret execution image references in terms of symbols defined in the source program [Kotok 61, Lane 73].

Some debuggers in use today are *source program debuggers*. In such debuggers the

machine representation is hidden from the user. They attempt to give the user a view of the program and its execution state in terms of the source code only. All references to program locations are expressed in terms of the source representation, i.e., lines in the text file. Data objects are displayed according to their type. Some expressions in the source language can be evaluated interactively [Mitchell 79, Unix 81b].

In LOIPE we provide a *language-oriented debugger*. Similar to the source program debugger, the language-oriented debugger presents the user with a source program view of the program and its execution state. In the language-oriented debugger, however, the source program is represented in constructs of the supported programming language rather than lines of a text file. The program state is mapped into structures defined by the language, e.g., the current value of objects is represented just as the initial value is.

A language-oriented debugger can make use of the language information in the program tree in two ways. In the *execution image based* approach the debugger works directly with the executing program in the representation of the target machine. Through the use of information from the program tree and compiler generated symbol tables the debugger is able to present a source program view of the program and its execution. The alternative approach *is program tree based debugging. In this case the program tree is the primary representation* being manipulated by the debugger. Both the static program and the program execution state are represented in the program tree. Access to the program state in the execution image is provided through the same mapping information that is used for code generation. The program tree based approach has been chosen for LOIPE. In the following section we give some arguments for this decision.

3.1.1. Execution Image Based Debugging vs Program Tree Based Debugging

The execution image based approach to interactive debugging dominates the realization of debuggers for compiled languages. There are several reasons for this dominance.

- The first reason is historical. The development of debuggers originated in the octal (or machine code) debugger. In this debugging system the execution image is the only program representation. Through the use of relocation information and mapping information between the execution image and the source, symbolic and source program debuggers are able to express program location references and data object references in terms of the symbols and representation of the

source program. The functionality of these debuggers, however, is still strongly modelled after the machine code debugger.

- The second reason lies in the need to debug programs that are translated with optimizing compilers. This is a difficult task if debugging at the source program level is desired. We are aware of only one attempt to approach this problem [Warren 75]. The commonly used solution is symbolic debugging support, e.g., Six12 [Lane 73] for the highly optimizing Bliss11 compiler [Wulf 75]. The debugger presents the user with an assembly code version of the program. It is for the user to determine the relationship between that program representation and the source representation.

- The third reason is that the source representation of the program is commonly maintained in text files. The logical structure of text files has no relationship with the logical structure of the program. Therefore, all references to the source program are on a line by line basis.

With the availability of the program tree as the permanent source program representation, execution image based debuggers can express program locations in terms of the logical program structure by mapping an execution image reference to a reference into the program tree. This is a first step towards language-oriented communication with the user. References to the source program are given in a language-oriented manner, but the debug functions are issued by explicit commands. In this *functional* approach to user interaction, each activity is performed through a separate command. Examples are separate commands for examining data object values, modifying them, setting breakpoints, or setting trace points. Invocation by command is necessary, because the textual representation of the program state and debug state information is generated directly from the execution image, thus cannot be manipulated by the structure editor. As a result the user interaction during program debugging differs from the user interaction during program construction.

The execution image based debugging approach must support two transformations between representations. On one hand, information from the execution image such as program state must be mapped into a textual representation for display. On the other hand, expressions that are supplied by the user as part of a debug condition or for evaluation, are in text or program tree form. Debuggers usually interpret them with respect to the execution image, i.e., an interpreter must be supplied as part of the debugger.

Program tree based debugging, in contrast to the execution image based approach, permits program construction and program debugging to be coupled tightly, resulting in a well-integrated programming support system. The user deals with a uniform interface, in that

both program construction and debug actions are issued through construction and edit commands. As mentioned earlier, both the debug state and the program state are included in the program tree representation and are manipulated through structure editing commands. The result is a form of interaction between the user and LOIPE that is referred to as *data driven* interaction [Sandewall 78], because system actions are caused by manipulation of data. The user does not perceive any difference between the programming and the debugging activity. The choice of the program tree as the primary program representation for debugging makes it the central program representation of the LOIPE system. This simplifies the structure of the LOIPE debugging facility. Only one mapping from the program tree representation to the execution image has to be performed. For this purpose the code generator, i.e., the mechanism that initially determines the mapping, is used. In chapter 5 we discuss how a consistent view of the execution image and the program tree representation is maintained for program state and debug state. In the remainder of this chapter we assume the state information to be available in the desired program tree representation.

The integration of the debug state and the program state into the source program representation requires extensions of its logical structure. This means extensions to the programming language, and extensions to the permanent source program representation in LOIPE - the program tree. *Extensions to the programming language have side effects.* The language may become more complex due to increased number of language constructs and concepts expressed by these constructs. The second side effect is that programs written in the extended language are not supported by existing programming environments for the standard language, thus are not portable unless the support system for the extended language is ported, too. We maintain that in the context of LOIPE language extensions have different effects on the programming environment. The language that is used in LOIPE to express the textual representation cannot be equated with the programming language that is used in traditional programming systems. In the following we point out the different meaning of the term programming language in the LOIPE context and argue that these extensions reflect the integration of the debug command interface into the program tree representation.

3.1.2. Source Program Representations and Programming Languages

In LOIPE programs are maintained in form of program trees and textual representations are generated dynamically by the structure editor. The structure editor is able to generate several textual representations with different concrete syntax from the same progam tree representation. For example, certain programs can be shown to the user with Pascal keywords or with Ada keywords. Does this mean that the two textual representations are expressed in different languages? The logical structure and the semantic meaning of the program are not changed, because the abstract syntax and the semantics associated with the elements in the abstract syntax representation, which determine the characteristics of a programming language are the same. Thus, in the context of LOIPE, two programs, originating from one program tree, but differing in the concrete syntax, are considered to be represented in the same abstract language.

The selective view mechansim of the structure editor (see section 2.1) permits LOIPE to suppress all extensions to the program tree (and language) that reflect the program state and debug state. Thus, LOIPE is able to produce a textual representation of the user program in the basic programming language. Thus, LOIPE does not limit the potential portability of programs.

The structure editor can also generate a textual description of program module interfaces and their interconnections as well as a textual representation of the implementation of modules from the same program tree, thus showing the program at two levels of abstraction. The notations used to describe the two levels are often considered two separate, but often closely related languages in traditional programming systems, e.g., Mesa and C/Mesa [Mitchell 79]. In these systems, the two representations frequently must be maintained separately by the programmer and require consistency checking of the represented logical information.

The LOIPE approach permits integration of the logical information in the abstract syntax representation. Duplication of information can be avoided in the permanent program representation, eliminating consistency checks of the redundant information. The concrete syntax of the two textual representations can be defined, such that the programmer does not perceive a difference in the notation used to express the two descriptions, especially the

redundant parts. The result is a uniform logical structure of the permanent source program representation and a consistent view of the different textual representations that are dynamically derived from the program tree representation.

3.1.3. Language Extensions - A Command Interface

At first sight the extensions to the language may seem confusing to the user. However, it must be remembered that the extensions do not increase the complexity of the notation in which the user program is written. These extensions are used to represent debug and runtime information, that otherwise would be shown to the user in a separate notation. The manipulation of this information through the structure editor has the effect of issuing debugging commands. A separate command interpreter is therefore not necessary. In other approaches such operations are applied with explicit commands. This increases the number of commands, requiring the choice of obscure keyboard sequences to invoke a command (see for example the emacs editor [Gosling 81a]).

In LOIPE the user manipulates the information through structure editing commands. Information is added, changed, or renewed. At any time the structure editor limits the applicability of commands that can be issued by the user. Certain pieces of the program tree can be shown read-only, i.e., cannot be modified through editor commands [Medina-Mora 81]. Furthermore, the availability of constructive commands is limited to a small legal set, enforcing the syntactic correctness of the program tree structure. For example, breakpoints cannot be set in a type definition. Thus, LOIPE provides the user with guidance as to which commands are sensible at any given time.

3.1.4. Summary on Integration of Language-Oriented Debugging

LOIPE provides a language-oriented debugging facility based on the program tree as its primary program representation. Both debug information and program execution state are integrated into the program tree representation. Not only the source program, but also the debug and program state are manipulated by the structure editor. The action routines that are invoked by the editor cause the debug function to be performed. The result is a simplified and uniform user interface in which only editor commands are applied. Interactions that are performed in other systems through special commands, are expressed in terms of manipulation of data.

By basing the implementation of the debugging facility on the program tree, LOIPE can take advantage of the language knowledge embedded in that program representation, and improve the functionality of debugging facility over that of existing source code debuggers. In the next three sections we discuss issues related to each of the three parts of the LOIPE debugging facility, the presentation of the program state in a language-oriented manner, the realization of debug operations in terms of language-oriented debug statements, and the user's control over the actual program execution.

3.2. Program State

The program state consists of the current values of data objects and the call chain of active procedures. Usually the user desires to examine the program state when execution is suspended, to monitor the progress of the execution, and to make changes to the program state. Such facilities have been the heart of debuggers for a long time. However, different debugging systems provide them in different form and with varying functionality.

In traditional debugging systems, the user must request the display of items in the program state, e.g., the procedure activation stack (callstack) or current value of a variable explicitly. The presented information is commonly displayed in the form of a scrolling script. Some systems support display of program state information in different areas (windows) of the display screen [Mitchell 79, Petit 69]. Data objects can be monitored by repeated automatic display at preselected program locations, e.g., procedure entry/exit [Lane 73], or conditional and unconditional user defined trace prints [Unix 81b]. A display request is issued in a functional manner, naming the object to be displayed. Examination of linked structures of dynamically created objects requires a name path to be specified, that indicates the full access pattern starting from a named object [Mitchell 79]. The values of objects are displayed in terms of the representation of the base types in the language [Unix 81b] and sometimes in terms of the user defined abstractions. The value of an object is usually changed through an explicit command or the evaluation of an assignment by the debugger's expression evaluator. The state of the control flow cannot be modified.

With the availability of high-resolution displays as part of a computer work station [Thacker 79] new ways of displaying information in an interactive debugging environment have been explored. The DLisp system [Teitelman 77]utilizes multiple overlapping windows on the same display. These windows are used to organize displayed information and to permit interaction

with different processes. It is possible to display linked data structures in graphical form. Due to the extensible nature of Lisp the user can add display routines of his own. Incense [Myers 80] is the frontend for an interactive debugging system for the compiler-based language Mesa. Similar to DLisp, this system supports multiple overlapping windows and utilizes the graphical display capabilities for analog representation of data structures. The analog display of linked structures permits the user to conveniently examine elements in that structure by selection with a pointing device. Information regarding the logical and physical structure of data objects is made available by the compiler in an extensive symbol table. Documents are associated with data objects in order to display them. A document describes the concrete display representation of a data object - similar to the unparse schemes in ALOE. Incense permits the user to define additional documents for the display of objects, and dynamically change between them.

LOIPE's debugging facility is display-oriented in that it makes use of the ability to divide the screen into different windows. All information is displayed to the user through the unparse mechanism of the structure editor. Currently, the unparse mechanism is limited to character representations. Graphical display would require an extension of the set of operations supported by the unparser and appropriate display hardware. Because the syntactic details of the textual representation can be adjusted in the unparse schemes for readability and for consistency with the supported language, we do not consider the exact textual representation as relevant as the logical structure of the program tree for the discussions in the remainder of this dissertation. We continue by discussing the representation of the state of control flow in the program tree and its visualization to the user in LOIPE. Then, LOIPE's support for the examination of the state of data objects in a language-oriented manner is elaborated.

3.2.1. Control Flow Display

The control flow state of an executing program consists of the stack of currently active procedures or *callstack*, and a *current execution point* in each of these procedure invocations. The current execution point refers to the callsite to the next procedure in the active procedure stack or to the point of suspension in the procedure on the top of the stack. The current execution point refers to a program location and can be shown by associating a program cursor with the appropriate subtree in the program tree. Similarly, an active procedure can be displayed by showing the textual representation of the program subtree that represents its definition. The actual stack of active procedures, however, cannot be

represented in the basic program tree of the user program. The stack may contain several invocations of the same procedure.

The callstack is represented by an additional structure in the program tree, a list of nodes. Each node corresponds to a procedure activation on the callstack, and has two components. One component is a reference to the definition subtree of the active procedure. The other component is a program tree reference to the current execution point. The position of the activation record of a procedure in the actual runtime stack can be derived from the position of the corresponding node position in the list.

The program tree structure representing the control flow state is automatically maintained by LOIPE. It is updated any time the execution image of the user program temporarily suspends execution. This may be the case for actual suspension of execution (breakpoint) or for tracing of control flow or data flow. Once the program tree representation has been updated, the control flow stack is displayed without explicit request from the user.

Two windows are used to show the control flow state to the user, a *callstack window* and a *control flow monitor window*. The callstack window shows the list of currently invoked procedures by their name, whereas the control flow monitor window is used to display one of the active procedures and its current execution point. The callstack is displayed in the following way. The rootnode of the stack of active procedures is associated with the root of the callstack window. This subtree is unparsed by showing only the component of each node in the list, that refers to the procedure definition site, and displaying only the procedure name from the procedure definition subtree. Actual parameters of an active procedure are displayed only if a display request is issued (see next section).

The cursor of the callstack window is used by the programmer to indicate the *current context* for other debug functions, e.g., examination of local variables and actual parameters. The current context is indicated by moving the cursor to the appropriate element in the callstack and selecting that element. Selection is performed by moving the cursor *down* at the selected element, which causes a action routine to be invoked with the editor action *faildown* [Medina-Mora 81]. This action routine causes the control flow monitor window to be updated to show the definition site of the selected active procedure and that window's cursor to be placed at the current execution point of the procedure invocation. The resulting display on the screen is shown in Fig. 3-1. By default the top element of the callstack is selected as

the current context when the control flow state is updated in the program tree. The effect is that the cursor in the control flow monitor window acts as the "program counter", showing the progress of execution.

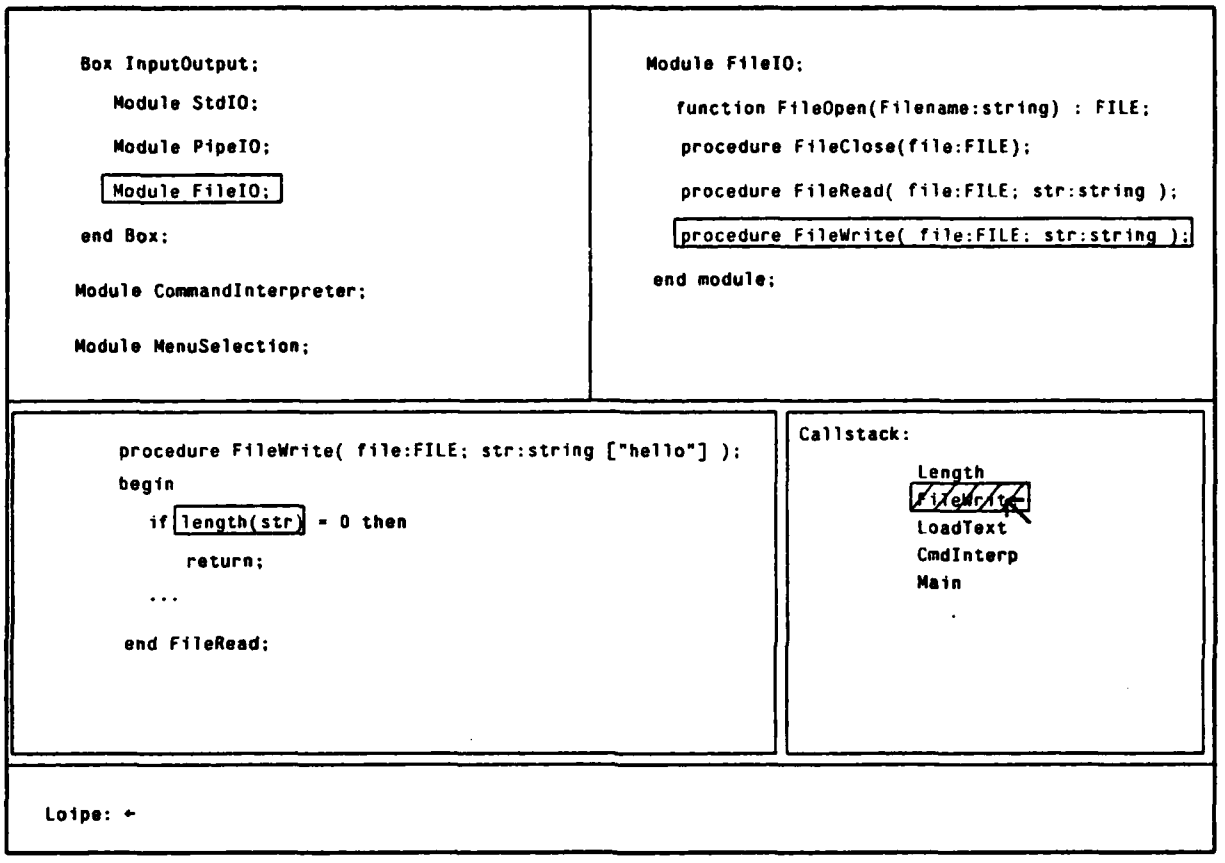


Figure 3-1: Callstack Display on Screen

The user has the ability to delete sequences of elements from the callstack. The effect of such a shrinkage of the callstack is that the current point of resumption is reset to the current execution point of the new top element on the callstack. In section 3.4 LOIPE's support facilities for controlling the excution of the user program are discussed in more detail.

3.2.2. Data State Display

The execution state of data is comprised of the current value of global or statically allocated variables, of local variables and local parameters that are allocated as scopes are entered, and of data objects that are dynamically created on a heap. The user is able to examine the data object state using syntax of the programming language and the names of abstractions defined in the program.

In LOIPE the data object state is represented as part of the program tree representation. Because the data object state is actually changed in the execution image during program execution, the program tree representation must be updated repreatedly by extracting the information from the execution image. However, only those parts of the data object state that have been requested by the user for display are updated in the program tree. The result is demand-driven display of data objects via the program tree representation and retrieval of their state in small steps. The mechanism for extracting data object state from the execution image will be discussed in chapter 5. We continue by first discussing the form in which data object state is represented in the program tree and as text, then elaborating in which way display requests for data objects are issued. Finally, the facility for modifying the current value of data objects by editing is introduced.

3.2.2.1. Display Format

The initial value of a data object is a special case of the current value. Thus, it is natural to represent the current value and the initial value in the same manner. Because many programming languages already provide support to specify the inital value of a data object as part of its declaration, we adopt the same representation for the current value.

Object values of basic data types are given in their standard representation. For objects of enumerated types the user defined value representation is used. In case of record structures both the name and the value of each component are given. An example of such an initial value representation for records is the record aggregate in Ada [DoD 80].

In the program tree representation the current value takes the following form. The current value subtree is a new offspring in the node that representation declaration. This subtree is constructed automatically by LOIPE when the declaration is entered in to the program. From the object declaration site information about the object and its type are directly available to

LOIPE. The definition site of the type is reached quickly by following the semantic link from the symbol table entry to the subtree of the type definition. The type information is used to determine the structure of the current value subtree. The current value subtree acts as placeholder, which is used to extract the current value from the execution image (see section 5.2.2).

For objects with a single value, i.e., objects of base types, and enumerated types, one node acts as a placeholder for the value. The unparsing mechanism of the structure editor converts the internal representation of the value into a textual representation, using the print routines that are normally provided as part of the runtime system of the language. For arrays and records the current value subtree is a list of nodes with two offsprings. One of the offsprings represents the name of a component in the record. The other offspring is the placeholder for the component value. Information about the record components is retrieved from the record definition site (Fig. 3-2).

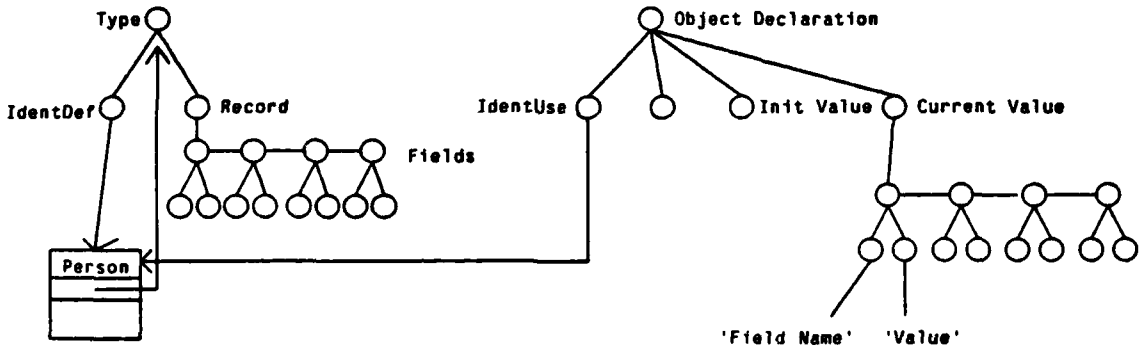


Figure 3-2: Current Value Of A Record Object

A component value can be a single-valued component, i.e., of a base type, subtype or enumerated type, or a nested aggregate of values, or a pointer reference. The display of a single-valued component has already been described. For nested value aggregates, a limited view is provided. The user sees nested aggregates only to a certain depth. For any component at that depth that is of record type the current value is shown as ellipses. Cursor movement down at the ellipses invokes an action routine with action *faildown*. This routine extends the display depth of a given data object, resulting in the expansion of the current value subtree. The value of a pointer reference is never shown, because it should be of no concern to the user. Instead, a special symbol or key word, e.g., '->' or 'ptr', indicates the

presence of a pointer in the actual representation. As in the case of the ellipses the user expands the display by moving the cursor down at the pointer field.

3.2.2.2. Data Display Requests

During program construction and usually during program debugging, the unparser selectively hides the current value subtree at the declaration site. Only those current values of data objects whose subtree is being displayed are retrieved from the execution image and updated in the program tree. The user can issue a display request for global data objects by changing the unparse mode of the window showing the declaration site of the global object to be examined. A change of unparse mode means that a different unparse scheme is associated with the generation of the textual representation in the given window. The newly selected scheme includes the current value subtree as part of the textual display.

Local data objects can only be displayed during their lifetime, i.e., if an active procedure invocation exists. In LOIPE, the display of local objects is caused as a side effect of the selection of the current context in the callstack. The actual unparse mode of the window showing the definition site of the selected procedure is changed to display the current values. The result of this display request is the extraction of both the parameter and the local object values corresponding to the selected context from the execution image into the program tree representation and their display or the screen.

The user can also examine linked data structures, i.e., dynamically created data objects, that are connected by pointer reference. The examination must start from a named pointer reference, i.e., a pointer or access variable or a record component that is of pointer type. Since linked data structures may be quite expansive, a separate window, the *data object examination window*, is provided for the display of such structures. The user follows elements in the linked structure by expanding the appropriate pointer references through cursor movement (Fig. 3-3). The user's ability to traverse the linked data structure by moving with the cursor through the current value subtree eliminates the common problem of having to specify the full name path from a program variable to the desired object. For linked data structures that represent trees or graphs it may be advantageous to show them in graphical form because the textual representation shows the links only in a limited scope. For a discussion of effective way to display interconnections using graphics, we refer to the work in Incense [Myers 80].

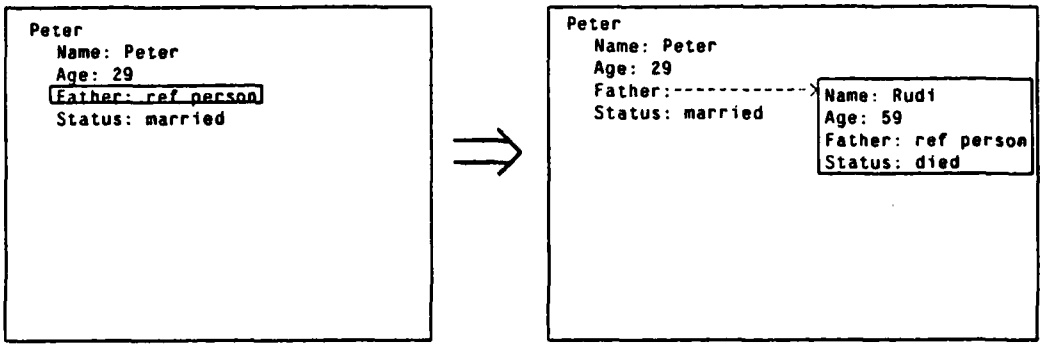


Figure 3-3: Examination Of Dynamic Objects

LOIPE supports *visual monitoring* of data objects. In contrast to dynamic assertion checking, which will be discussed in section 3.3.2, visual monitoring continuously displays the current value of data objects and leaves it to the user to detect any anomalies in the values. The current values of all data objects being monitored are shown in a *data object monitor* window. This avoids switching of the program window to the declaration sites of the different data objects to be displayed. Monitoring of a data object is requested by entering the data object, i.e., its name, into the data object monitor window. The effect of this editor operation is the establishment of a crosslink from the subtree in the data object monitor window to the declaration site.

The current value of monitored data objects is refreshed in place any time execution is suspended temporarily. Thus, the user can define a set of data objects to be examined whenever a break point is reached or an assertion does not hold. The effect of monitoring all modifications of a data object can be achieved in connection with the ability to associate the trace action with data object declarations (see section 3.3.3).

3.2.2.3. Modification of Data Object State

LOIPE provides two ways for changing the current value of objects. The expression evaluation facility (described in section 3.4.1.1) processes statements with the effect of an assignment statement changing the current value of the assigned variable. The more common way of modifying the current value, however, is realized through the ability of the structure editor to manipulate the program tree structure representing the data object state.

The user can move the cursor to the node holding the current value and apply modification operations. For record aggregates only the component values are modifyable by the user. This is enforced by the structure editor. The display of the component name is specified read-only in the unparse scheme, i.e., is not accessible by the cursor. Deletion at the root of the current value subtree is prevented through the access control mechanism in ALOE [Medina-Mora 81].

Pointer values can only be modified by expression evaluation. In that case all the semantic rules of the context in which the evaluation is performed apply. Thus, the user cannot do more damage than he could through the program itself. In the program tree representation for current values pointer values are maintained, but never shown to the user (see paragraph 3.2.2.1). The special symbol, that is shown instead, is not modifyable.

3.2.3. Information Hiding for Record Types

Progams that are written in languages with information hiding through software modules, a record type may have two specifications. One is the complete definition of the record type in the module providing the implementation. The other is the exported definition for use outside the providing module. This definition may hide some of the component details. This brings up the question as to whether the declaration of a record type object outside the defining module should be bound to the exported record definition site, or to the implementor's definition site for the purpose of generating the current value representation. LOIPE can realize either one of the two approaches, since the necessary semantic information is available.

From the pratical point of view, the limitation of data object display to the accessibility defined in the export definition is too restrictive. The user could examine only those parts of a data object that are visible. For display of the full content, execution would have to be suspended in the implementing module and the data object would have to be accessible at the point of suspension. Therefore, LOIPE supports examination of data objects according to the full definition. The user is expected to be cooperative in that he is aware of the potential damage to the consistency of a data object through modification of the data object state.

3.2.4. Summary of Program State Representation

LOIPE supports access to the progam state in a language-oriented manner. The program state is represented as part of the program tree. This representation allows the user to examine and modify the program state in the same manner the source program is manipulated. The program tree representation of the program state acts as a write-through cache for the actual program state in the execution image. The current value of program state parts are extracted from the execution image into the program tree upon display requests. Changes to the program tree representation are recorded in the execution image immediately.

3.3. Debug State

The debug state of a program records the definition of debug statements, the locations of their application, and whether they are enabled. Debug statements may be unconditional debug actions, such as breakpoint or tracepoint, or conditional debug actions with user-defined conditions.

In traditional programming systems, the debug state is represented and maintained in (a combination of) two ways. User-defined debug statements are expressed through the supported language, e.g., conditional point statements, and are enabled through the conditional compilation mechanism. The enabling of such debug statements requires recompilation and linking, thus, cannot be performed interactively. Interactive debugging support is provided through a separate debugging tool. This tool is used in conjunction with the executing user programs. Through commands, the user interactively defines and enables debug statements in a form that is accepted by the debugging tool. The debugging tool records the debug state in a structure that is separate from the source program and modifies the execution image of the user program to reflect enabled debug statements. The debug state is lost when leaving the debugger, because the state is maintained only within the debugger. The debugger, however, must be left in order for the user to make modifications to the program.

Debugging systems usually support breakpoints and tracepoints. The user may specify a condition under which these debug actions are applied. One kind of conditional debug statement, the *assertion*, has originally been included in a program as information for formal

program verification. Assertions specify conditions that are assumed to hold for a certain program state. The inclusion of assertions in the program text documents the expressed assumptions. It has been recognized that dynamic checking of these assertions provides an excellent debugging aid [Satterthwaite 75, Lampson 77]. Assertions have the property that they do not affect the program state of the user program when enabled. This means that user programs can be executed with enabled or disabled dynamic assertion checking, without changing the program behavior (except for execution speed). Assertions without side effects can be guaranteed by restricting the assertion condition to expressions without side effects [Lampson 77]. The restriction of the assertion condition, however, has been recognized as too limiting for the expressive power of assertions and extensions have been provided such as quantifying operators and access to previous values of variables [Deutsch 73, Martin 77].

When defining a debug statement the user not only specifies the debug action and the condition under which the action is taken, but also the program location at which it is applied. In the most common case a debug statement is defined for one program location, e.g., a breakpoint is set at one location. Some debugging systems allow a range of program locations to be specified for a given debug statement. Examples are single stepping, i.e. breakpoint at every instruction of the whole program [Kotok 61], array bound checking for all array accesses [Jensen 74], break or trace for all procedure entries and exits [Lane 73], and evaluation of an assertion condition in a specified program region [Martin 77].

In LOIPE debug statements support debugging activities that range from simple breakpoint to dynamic assertion checking and execution timing. Debug statements are maintained as part of the program tree, thus, are documented permanently and are not invalidated by program modifications. Debug statements are defined to be applicable within a certain scope, i.e., at a range of program locations. The scope is determined by the scope rules of the language. In LOIPE, debug statements are enabled for runtime evaluation independent of their definition. This separation of definition and enabling of debug statements permits an enabling statement to be associated with several debug statements. The separation, more importantly, permits debug statements to be defined permanently in the program without being enabled. The effect is the documentation of test conditions for later use.

In the remainder of this section we discuss LOIPE's support of debug statements in more detail. First, the semantics of debug statements are given. Then, the dynamic assertion checking mechanism in LOIPE, whose expressive power supports all assertions handled by

existing systems, is presented. Finally, the scope of applicability of debug statements and the resulting debug activities, and the mechanisms and cost of enabling debug statements are discussed.

3.3.1. Semantics of Debug Statements

A debug statement basically consists of a debug condition and a debug action. If a debug statement is enabled for runtime execution, the condition is evaluated to determine whether the debug action should be performed. LOIPE provides two predefined debug actions: *execution suspension*, and *execution tracing*. Execution suspension means that the executing user program relinguishes control to LOIPE when this debug action is encountered in the control flow. LOIPE, then, displays the program state and accepts user interaction. If the executing program encounters an execution tracing action, execution is temporarily suspended to display the program state, but does not prompt for user interaction.

Debug statements can be dis- or enabled without changing the program behavior (other than execution speed). By attaching strict semantics to debug statements, LOIPE can statically enforce that this is the case by ensuring that the debug statement does not perform write access to a data object in the program. Without this restriction the expressive power of debug statements is increased, but application of debug statements may change the program behavior. The enabling mechanism for debug statements would effectively become a conditional compilation mechnism.

If side effects are permitted in the debug action, the debug statements take up the characteristics of an exception handling facility. Violation of the debug condition raises an exception - see for example range-error or assert-error in Ada [DoD 80]. The debug action acts as an exception handler that is statically bound to the exception. The handler may examine and modify the program state. Upon completion, execution is resumed at the location that raised the exception. An exit or return statement in the debug action achieves the effect of exception handlers in Ada, namely abortion of the program unit in which the exception was raised.

In the context of LOIPE, we assume that both conditional compilation and exception handling are provided through separate mechanisms, and limit ourselves to the stricter semantics for debug statements. The three mechanisms fulfill different tasks and, therefore,

should be available to the user as distinguishable mechanisms. Conditional compilation is commonly used to maintain different versions of programs in one source file. Exception handling provides the capability of handling recurrent exceptional conditions permanently as part of the expected user program behavior. Debug statements facilitate monitoring of the program state for unexpected behavior. The user is informed of such an occurrence, but the cause of this unexpected behavior is assumed to be remedied by modifications in the user program.

3.3.2. Dynamic Assertion Checking

LOIPE supports dynamic checking of assertions that contain quantifying operators and references to previous values. If quantifying operators are not provided by the programming language, the user can emulate the effect of the quantifying operators by a value-returning function without side effects (e.g., a Euclid function). This function can contain loop variables for stepping through a set. A natural extension is to permit the declaration and use of local variables in the debug statement directly rather than requiring the user to define a separate function for that purpose. This is achieved by using a value-returning block with write access to locally declared data objects as assertion condition.

In verification *previous values* of data objects are often referenced in post conditions for procedures (to specify how the program state relates to the program state at procedure entry) and in iteration and recursion invariants (to make claims about the progress of execution). It is difficult to come up with a general rule that determines when the content of a data object should be saved for later reference as previous value, such that it can be done automatically by the system. The previous value may refer to the value at procedure entry, the value at the last execution of an assertion, or the value before the last write access to the data object. Therefore, we follow the route taken in the Interactive Program Verifier [Deutsch 73] and the HAL/S testing system [Martin 77], i.e., require the programmer to specify when and where the old value of a data object is saved. For that purpose we introduce the notion of debug variables in LOIPE.

Debug variables have a scope that is larger than a single debug statement. They are declared in the same manner as program variables, but are marked as debug variables. The actual choice of textual representation to distinguish the declaration of debug variables (as well as the definition of debug statements) is irrelevant at this point. For example, in this

dissertation, we use the keyword *assert* to visualize the difference between elements of the user program and elements of the debug state.

Program variables and debug variables have the same scope rules, but differ in that debug variables are only accessible in debug statements (in the latter only with read access). Debug variables can be modified in both the debug condition and the debug action. The modification of debug variables does not violate the semantics of debug statements because debug variables are inaccessible by the user program, thus not part of the program state. The abstract syntax description for debug statements must take into account that debug variables can be modified as a debug action, i.e., assignment statements must be permitted in addition to the break and trace actions. The full abstract syntax description of debug statements is given in appendix A.2.

With debug variables, the user can maintain previous values of program variables as necessary to express assertion conditions. With this quite powerful assertion mechanism, the user can monitor relationships between data objects and between time sequenced values of data objects at a high level, i.e., the level of abstractions provided in the program. The ability of debug variables to maintain state information for debugging purposes can be used for performance measurement. For example, frequency counts and time measurements may be kept in debug variables. They can be used for statistical processing or displayed. A discussion of the full potential of performance monitoring support, however, is beyond the scope of this dissertation.

3.3.3. Scope of Debug Statements

Debug statements have a *scope of application*. This scope defines the range of program locations at which a debug statement would be evaluated if enabled. Usually, the definition site of a debug statement corresponds to its application site, i.e., debug statements are defined in the code only. However, [Taylor 80] has recognized that there is a need for defining a debug statement once and specifying a scope in the program for which it is applicable. It is commonly the case that certain assertion conditions should hold for a program part, e.g., a procedure or module. The assert facility in Euclid [Lampson 77] requires the user to repeat the assert statement at all application sites. The Hal/S system [Martin 77] permits the indication of a range of statements for which the debug statement is to be applied. A preprocessor takes care of the replication of debug statements.

In LOIPE the scope of a debug statement is determined by the location of its definition. If the debug statement is defined in a statement sequence, the definition site is the location of application. If the debug statement definition is associated with a program block, procedure, or module the scope of applicability corresponds to that defined by the given language construct. Debug statements can also be associated with data objects or data types. In both cases, the scope of applicability is that of the respective item.

As a combination, the expressive power of debug statements and their scope of application provides a rich debugging environment that should satisfy the most common needs of programmers. For example, the unconditional debug action *break* associated with a procedure provides single-stepping through that procedure. The break action does not apply to other procedures being called by the target procedure if not applied in their scope. Unconditional trace action associated with a module has the effect of showing the progress of control flow in that module with the structure editor cursor. Conditional debug activities or assertions, when defined for a procedure or module, make sure that the assertion condition holds at any time within that scope.

The association of a debug statement with a data object declaration causes the debug statement to be applied whenever the data object is modified by the program. If the debug statement is an unconditional trace action, the result is monitoring of the assert object value. Assertions associated with a data object will guarantee the validity of its condition throughout the lifetime of the object. Assertions can also be defined for a data type. In this case, a statement is made about any object of the given type.

3.3.4. Enabling of Debug Statements

Once a debug statement is defined the user can choose to enable it for execution. The user may want to enable a single debug statement definition, or all definitions of debug statements within a certain scope, or even all debug statements that have an application site in a given scope. LOIPE provides two mechanisms for doing so. The first mechanism allows the user to enable individual debug statements by marking the definition site appropriately. The definition site can be marked in three ways.

1. The state can be recorded in an additional offspring to the debug statement node. The user may fill the binary state value by constructing a node of one of two legal terminal productions. The offspring is shown textually as a keyword (see Fig. 3-4.a).

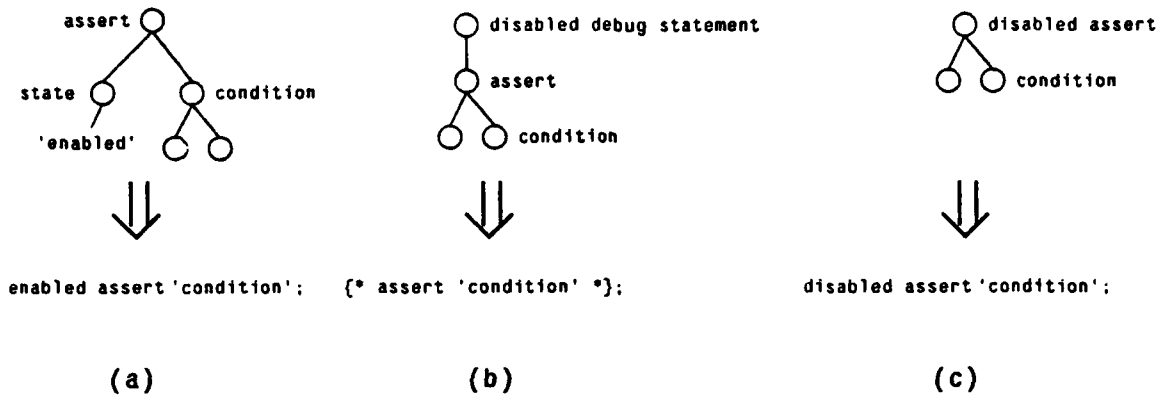


Figure 3-4: Alternative Tree Representations For State Information

2. The state can be recorded by the existence of an additional ancestor node. The enabled debug statement is represented by the usual program subtree. However, if disabled, the debug statement subtree is nested into a node which acts as the disabling indicator. In textual representation, the act of disabling may appear to the user as nesting the debug statement into a comment. This alternative is illustrated in Fig. 3-4.b.

3. The state can be represented by two productions in the abstract syntax *description for the debug statement node. The* two productions have the same offsprings. They differ, however, in that the semantic interpretation of one means disabled debug statement, whereas the other indicates an enabled debug statement. The application state is changed by applying the *transform* command of the structure editor to the debug statement node in order to convert it from one node type to the other. The textual representation may show the state either as keyword or bracketing symbols (see Fig. 3-4.c).

The implementor of a LOIPE, i.e., the person providing the grammatical description of the programming language, decides the way of marking by specifying the approapriate productions in the language description. We suggest the use of a separate state node because the editing commands to change the state are most obvious to the user.

The second mechanism for enabling debug statements in LOIPE is provided through an enabling construct. The *pragma* statement in Ada [DoD 80] is an example of a construct that provides directives for the programming system. Similar to the definition site of debug statements, the location of the enabling construct in the program tree determines the scope for which it is applicable. By providing the enabling statement with a procedure name, for example, the user indicates that all debug statements with an application site in the given

scope whould be enabled. This includes debug statements where scope of application conforms the enabling scope as a subset.

3.3.5. Cost of Debug Statements

Debug statements and enabling constructs are maintained by the user in the program tree representation. LOIPE, however, does not include debug statements in the execution image unless they are enabled for runtime evaluation. Thus, no cost penalty incurs on the executing program for unused debugging aids. This approach requires processing by LOIPE to include debug statements in the execution image when they are enabled. For debug statements that are enabled by modification of the application state the processing cost is proportional to their scope. For example, enabling of single stepping in a procedure requires processing of that program piece. For conditional debug statements, processing is restricted to those program pieces in the scope of applicability whose execution may affect the condition being tested. For example, an assertion associated with a global data object requires processing of those procedures that actually access the object for modification. These program pieces can be determined without much difficulty from the available semantic information in the program tree.

The processing cost directly affects the responsiveness of LOIPE as an interactive system. Therefore, LOIPE makes the user aware of the processing cost for the request to enable debug statements. In order to issue the enabling request through an editing command the user must move the editor cursor to the definition of the debug statement or to the enabling construct. As the cursor moves through the ancestor node, it gives a good estimate of the scope of applicability by highlighting the appropriate program piece, e.g., a procedure or a module.

3.3.6. Summary of Debug State in LOIPE

By integrating the debug state into the program tree representation, LOIPE extends the uniformity of user interaction through editing commands to the definition and enabling of debug statements. Debug statements are entered and modified in the same way as the user program, i.e., in a language-oriented manner. By separating definition and enabling of debug statements, LOIPE permits the user to document assertions as part of the source program for later use without incurring any cost on the executing program. Because LOIPE's debugging facility uses the program tree as its primary program representation, it can make use of the

available language knowledge and semantic information. One example is the capability to associate debug statements with various constructs in the language, thus, specifying different scopes in which the debug statements would be applied. By exploiting the limits of the semantic restrictions for debug statements, LOIPE increases the expressive power of debug statements beyond conditional break and trace statements. As some other systems LOIPE is able to support sophisticated dynamic assertion checking with quantifiers and previous values. In addition LOIPE permits assertions to be defined at different levels of abstraction. The debug statement mechanism can also be utilized for performance monitoring tasks.

The language-orientation permits LOIPE to increase its debug functionality with the power of the supported language. New concepts can be utilized by LOIPE's debugging facility, as they are included into languages. One such candidate are path expressions, which have already been included in an extension of Pascal [Campbell 78]. Path expressions define certain ordering relationships between operations on a data type. These are in their original form intended as a synchronization mechanism, but can also be interpreted as constraints on the application of those operations. In [Habermann 79b] a realization for dynamic path expression checking is indicated, which has been implemented in [Habermann 78, Andler 79].

3.4. Execution Control

Execution control gives the user the ability to influence the normal flow of control during program execution. Execution can be started and suspended. After suspension the user can continue execution or redirect it to a different program piece. This capability is essential to program debugging.

Execution control can be found in various forms in existing debugging systems. Usually the execution of a program is suspended if a hardware exception (e.g., divide by zero) occurs and is not handled by the user program. The exception is handed to the debugging system for reporting to the user. If the debugging system is not active the operating system is informed, which then aborts the user program and possibly saves a copy of the current execution image for post-mortem examination, e.g., Unix [Unix 81a]. Debugging systems permit the user to interactively define breakpoints. Suspension occurs when execution encounters such a program location. In many systems the user can interrupt the program execution by typing one or a sequence of special control-characters.

Once the user program is suspended the user can examine the program state and issue debug commands. Execution can be continued in several ways. Program execution can be started from scratch. Execution can also be resumed at the point of suspension, if the user interactions are limited to debug commands. Usually, after modifications to the user program, execution must be started over in compiler-based systems. Some debugging systems support interactive expression evaluation, that accepts as one of its expressions the invocation of a procedure or function. This allows the user to continue the execution at a different program location.

LOIPE is similar to other programming systems in the provision of execution suspension facilities. In contrast to other compiler-based programming systems, however, LOIPE supports the execution of incomplete programs. Execution is suspended when an incomplete or incorrect program piece is encountered. The location of execution suspension is shown through the structure editor (see section 3.2.1), and a message informs the user of the cause of suspension. Upon suspension of execution the user of LOIPE can freely mix manipulation of the user program, the program state, and the debug state. Some of the modifications may affect the ability to resume execution at the point of suspension. LOIPE keeps track of the user modifications and their side effects on the execution image, and determines whether the user may be permitted to resume execution, or may be required to continue at an earlier point in the execution history. The user can also continue by redirecting execution to a different program part. In the remainder of this section we elaborate on the mechanisms for continuation of execution, and investigate the criteria that are applied to determine whether execution can be resumed at the point of suspension.

3.4.1. Continuation of Execution

In LOIPE the user causes execution of the user program either by issuing an extended editor command, or by evaluation of an expression. The extended command *continue* is one of two commands that have been added to the structure editor for LOIPE. The continue command causes program execution to start at the point that is indicated by the program execution state. After a suspension of execution this is the point of suspension. When a program is being constructed the program execution state is defaulted to the entry point of the program and the initial global data object state. Thus, the initial continue command has the effect of starting the program - if the program entry point exists. At any time the user can reset the program execution state to the initial state with the second extended command *initialize* (see

paragraph 3.4.2.3). The command sequence *initialize continue* will start execution of the user program from the beginning.

The user can also start execution at a point that is different from the program entry point or the point of continuation indicated by the program execution state. This is accomplished through the evaluation of an expression or statement that is supplied by the user. In this program piece procedures and functions can be invoked, i.e., execution is redirected. The next paragraph discusses the program evaluation mechamism of LOIPE in more detail.

3.4.1.1. Program Evaluation

Program evaluation is the ability for the user to specify an arbitrary piece of program interactively and have it executed immediately in the current context of the suspended program. Upon completion control is returned to the point of execution suspension. Some compiler-based debugging systems provide such a facility under the term expression evaluation. The debugger interprets the entered expression and displays the result of the evaluation.

In LOIPE any executable program piece, i.e., expressions and statements, that can be expressed by the *supported programming language is eligible for execution if it is* semantically correct in the given context of the suspended program. The context is determined by the element selected on the callstack. The user constructs the piece of program to be evaluated with the structure editor in a special evaluation window. The user can also select the program piece from the program window and clip it into the evaluation window. Several program pieces may be maintained in the evaluation window.

When LOIPE is informed to evaluate a program piece it invokes the semantic checker on the selected subtree for the given context. If the subtree is semantically correct code is generated and loaded into the execution image. Then control flow is rerouted to start executing that temporary piece of code without destroying the runtime stack. When execution is completed the result of the execution, if any, is displayed, control flow is reset to the point of suspension, and the temporary code piece is removed from the execution image.

This mechanism is used for different purposes.

- Data objects can be examined by simply submitting their name to the evaluator. With the implementation described above the overhead for examination is

relatively high. However, LOIPE provides another facility for displaying values of data objects (see section 3.2.2).

- Individual fields of composite data objects and elements of linked structures can be accessed. Again, LOIPE already provides an efficient mechanism.

- Arbitrary expressions can be evaluated. This includes invocations of procedures.

- Evaluation of program statements may affect the program state. An assignment statement to a data object has the effect of modifying the current value, an alternative to editing the current value directly.

- The evaluation of the return statement differs from other statements, because it causes program execution to resume by returning from the top procedure invocation, possibly with a user specified return value. This allows the user to prematurely complete (abort) a procedure invocation.

The cost for the evaluation of an expression seems relatively high, because code is generated and loaded into the user program. However, the advantages of this approach justify its cost. In the LOIPE approach, code is executed on the target machine evaluating the expression using target machine arithmetic. The debugger subsystem has no specific knowledge about the target machine or the supported language. This knowledge is embedded in the semantic analyzer and the code generator. In the conventional (interpretive) approach to expression evaluation, mechanisms must be provided that understand all constructs that are legal in the supported language, know to interpret the runtime environment including accessing mechanisms, and simulate the arithmetic of the target machine, if it differs from the host machine. This machinery makes the debugging system both language and machine dependent, requires higher complexity of the debugging software, and often does not perform a complete job.

3.4.1.2. Unwinding of Execution Flow

The continue command allows the user to resume execution at the point of suspension. This is, however, only possible if the control flow state, i.e., the call chain of active procedures, is not affected by modifications to the user program. In some cases the control flow state can be adjusted to correct the affected part. LOIPE detects such effects on the control flow state. The detection criteria are discussed in the next section. When LOIPE has detected an affected control flow state, it *unwinds* the control flow state automatically to remove the damage. Unwinding of the control flow state means that the point of resumption is reset to the current execution point of the first undamaged active procedure. The data object

state is not restored to previous values. LOIPE informs the user of the correction to the control flow state such that the user may decide not to continue execution because of inconsistent data object state.

The user is also given the opportunity to unwind the control flow state explicitly. This is done by deleting the appropriate elements from the program tree represention of the control flow state (see section 3.2.1). Removal of all elements on the callstack and application of the continue command has the effect of restarting the execution without reinitializing the global data object state.

3.4.1.3. Restoration of Program State

Restoration of program state refers to the ability to reset the current state of program execution to be one at an earlier point in execution. In effect, the state is assumed to be the one as if execution had been suspended and restarted at the appropriate point. Restoration by starting execution over has several disadvantages. Repeatability of the execution must be guaranteed. This may be difficult if the execution includes access to external devices or files. Furthermore, the computational effort may be unacceptably high if long-running programs are involved. Therefore, we consider the alternative approach of restoring a previous execution state from saved state information for LOIPE.

We take a simplistic approach to restoration of execution state. We limit the execution state to be restored to the execution image, i.e., to the source program and program state. Furthermore, restoration is restricted to predefined points in the program. One of these points is the initial program state. LOIPE restores that state by totally unwinding the callstack and reinitializing the global data object state. Reinitializion of the global data object state consists of updating the current value in both the program tree and execution image to the initial value. Reloading of code is avoided. The user can request this restoration by the extended command *initialize*.

Other points of restoration must be defined by the user explicitly. A potential restoration point is indicated by a new language construct, *keep*. Whenever execution passes through such a point a copy of the execution image is saved in a file that is associated with the given restoration point. The program state can be restored to one of these restoration points if the enclosing procedure is active on the runtime stack. The restriction of restoration to points in active procedures is necessary to ensure that the saved execution image has not been

affected by any user modification. Otherwise, the saved snapshots of the program execution would have to be checked explicitly for side effects of user modifications. Due to this restriction the usefullness of user-defined restoration points is limited.

As the reader will have noticed the support mechanism for restoration is somewhat clumsy and inefficient. We have included it only to complete the set of mechanisms that aid in continuation of execution. Additional research will be required for the design of a general restoration mechanism that withstands the dynamic changes to the program representation in the LOIPE environment, possibly drawing from the experience with recovery blocks [Randell 75] and stable variables [Liskov 80].

3.4.2. Consistency of The Program Execution State

Modifications to a user program change its logical structure. Such a change may affect the semantic consistency of the user program, i.e., the modified part and all program parts depending on its semantics. The modification may also affect the structural consistency of the program execution state. *Structural consistency* of the program execution state refers to the validity of program state information with respect to the current instance of the user program, such that resumption of execution can be permitted. This means that the control flow state contains correct references to existing program locations, and the data object state reflects a valid state for all active data objects, i.e., all data objects whose lifetime includes the current point of execution resumption. The content of data objects is not part of the structural consistency.

In traditional compiler-based systems, the program execution state is maintained consistently only for modifications due to debug actions. For any modifications to the user program, the program execution state is discarded, the consistency of the execution image is restored by reconstruction, and execution must start from the beginning. In interpretive systems the user can make modifications to the user program as well as apply debug statements without loss of data object state. However, user program modifications may affect the control flow state. It is for the user to decide whether execution resumption is safe. The user can always redirect execution by invoking a function through expression evaluation.

In LOIPE the user the user does not distinguish between a modification of the user program and a modification of a debug statement. Therefore, LOIPE must be able to maintain structural

consistency of the program execution state for a class of user modifications that include the manipulation of debug state. As a result, resumption of execution is possible not only after debug interactions, but also after modifications of the user program.

Structural consistency of the program execution state can be maintained in several ways. Two are being used in traditional compiler-based systems. In those systems, debug statements are inserted into the executable code in such a way that the program state is not damaged. Resumption of execution can safely be permitted. We refer to this class of modifications as *non-damaging* modifications. All other modifications require restart of execution and are therefore classified as *fatal* modifications.

In LOIPE, we increase the number of *non-damaging* modifications by locating those modifications of the user program that leave the program execution state invariant. We also introduce two new classes of modifications; the *correctable* class and the *restorable* class. The correctable class contains modifications, for which the structural consistency of the program execution state can be corrected without changing the point of resumption. The restorable class consists of modifications, whose side effects do not permit corrections. However, the program execution state can be restored to a state for which structural consistency holds again. An extreme case is the restoration to the initial execution state. The next four paragraphs discuss the classification of modifications into the four classes and the criteria used by LOIPE to determine the class of a modification.

3.4.2.1. Non-Damaging Modifications

Non-damaging free modifications do not affect the structural consistency of the program execution state. Therefore, modifications do not require additional processing, once they are recognized as a member of this class. In two occasions a modification is non-damaging. A change to the initial value of a data object in its declaration does not affect the program execution state, i.e., the current value of the data object. The current value is only restored to the initial value when execution is started from the initial state or control flow enters the block containing the local declaration. Similarly, any change to a procedure that is not active, does not involve the control flow state or the local data object state. An active procedure can be recognized by checking whether it is a member of the callstack subtree structure. The class of non-damaging modifications contains a variety of modifications. Procedures and local variables can be added, modified, or removed, if the procedure (or enclosing procedure) is not active. In all cases it has to be remembered that a user modifications itself may be

non-damaging, but the propagation of semantic information may result in structural inconsistencies due to the change in the affected program part.

3.4.2.2. Correction of Program State

The corrections of the program state are limited to updating the control flow state, and certain changes to the data object state. The control flow state is corrected if the change involves an active procedure, but the current execution points of the procedure are not directly modified. A current execution point is modified directly if the callsite representing the current execution point is a member of the statement subtree, being modified by editing operations. The modification of such a subtree is excluded because it may have damaging effects on the program state (see paragraph 3.4.2.3).

The control flow state is affected in two ways. The current execution points may have a new position relative to the entry to the procedure, or the program part containing the callsite may have become nonexecutable. In the first case, the control flow state information in program tree representation is not affected because the references to the procedure definition site and the current execution points are expressed in terms of node references into the program tree. These node references are not changed by insertion or deletion in front or after a callsite location. This invariance property of the program tree representation permits LOIPE to correct the side effects in the execution image equivalent of the program state.

The second kind of effect on the control flow state, i.e., nonexecutablity of the callsite, is due to semantic errors or incomplete program pieces. If the semantic error is contained in the enclosing statement subtree, any correction of the semantic error by the user may damage the correct execution point. However, if the executability unit is chosen to be greater than one statement, e.g., a prodecure, it is desirable to correct the control flow state for the case that one of the other statements is the cause of the nonexecutability. This nonexecutability is frequently temporary and is followed by a user modification without damage to the current execution point. LOIPE corrects the control flow state by marking the current execution point. If, upon resumption, execution encounters such a marked current execution point, execution must be suspended.

Structural consistency is also corrected for addition or removal of global data objects. Creation of a new global data object causes its current value to be initialized to be its initial value. As a side effect of the creation previously undefined use of a variable may be bound, a

previously bound variable may be rebound to the new declaration, or a name conflict may have been introduced. Similarly, deletion of a global data object results in rebinding or undefined user of a variable. These side effects, if they occur, result in changes to program parts that fall into the classes side effect free, correctable or restorable.

3.4.2.3. Restoration of Previous Program State

If the program state cannot be corrected, it can be restored to an earlier state by disposing of the program state information in question. Removal of an element from the callstack has the effect of disposing of some control flow state as well as some of the local data object state. Thus, unwinding of the procedure activation stack provides the restoration mechanism (see section 3.4.1.2).

Modifications or side effects to statement subtrees containing current execution points, i.e., active callsites, are included in the restorable class. This has been done for several reasons.

- The modification may have mutilated or even removed the active callsite. The current execution point on the callstack has no meaning.
- The modification may have nested the active callsite into a loop statement or a block with local declarations. In such cases the state of local variables is undefined.
- In the semantics of many languages the the order of evaluation is not strictly defined within expressions or statements. Therefore, the actual order of evaluation is implementation dependent and resumption of execution may have unexpected results.

The occurence of a modification or side effect in a subtree containing an active callsite can be detected while the affected program piece is checked for semantic consistency. Every tree node, representing a procedure or function invocation, is checked for membership in the set of active callsites recorded in the callstack subtree. Once an active callsite is detected, the program state can be adjusted. All active procedures above the procedure activation containing the suspicious active callsite are removed from the callstack and the current execution point, i.e., point of execution resumption is reset to refer to the beginning of the affected statement. The effect of this restoration is that the call including the parameter setup is repeated. If the statement containing current execution point is deleted, the point of execution resumption reset to refer to the beginning of the succeeding statement. Thus, removal of a breakpoint, at which execution is suspended, is possible with the result of resuming execution at the next statement.

Unwinding of the procedure activation stack is a remedy for modifications that affect the local data object state. Addition and removal of local variables are included in this class in contrast to the addition or removal of globals for two reasons. First, the lifetime of a local data object is determined at runtime upon invocation of the enclosing procedure. At that point the current value is initialized. The creation of a local variable in an active procedure may leave the current value in an undefined state. For global data objects, the lifetime is determined by their existence in the program, i.e., they have current value, at the time of their creation. Second, the commonly chosen representation of local data objects in the execution image does not provide the flexibility to correct the data object state to include or remove local data objects (see section 4.2.1.3). Global data objects, however, are added or removed individually from the execution image through the partial replacement mechanism (see section 4.2.1.3).

Structural inconsistency, due to modifications that change the representation of a local data object, i.e., change of type in the declaration or modification of the type, can also be resolved by unwinding. Modifications to local data objects are recognized by determining from the context that a change has been made to the specification part of a local declaration. Once such a change has been determined, the enclosing procedure is checked for membership in the callstack subtree. Thus, modifications to local variables fall into the restorable class only if the enclosing procedure is active. Otherwise those modification belongs to the side-effect free modification class.

3.4.2.4. Fatal Modifications

Fatal modifications are those changes for which the program state cannot be preserved or restored, but must be set to its initial state. Thus, all modifications that affect the global data object state of both statically and dyamically allocated data objects, other than creation or deletion, belong into this class. Such modifications are replacement of the type of a global data object declaration or modification to a data type definition that is used in a global declaration. The current value of the data object, which exists under the old type definition, is not valid under the new definition.

In general, the representation of a data object cannot be converted automatically to the new representation without conversion instructions by the user. The maintenance of multiple versions of data object representations has been investigated in [Goulon 78] for continuously running systems. For program debugging in LOIPE, however, we do not consider such a facility as essential. A modification that changes the representation of data object

representation affects the semantic consistency of many program parts, which then must be corrected by the user, such that resumption of execution is hardly desirable.

3.4.2.5. Detection of Structural Inconsistency

A modification can be an explicit modification by the user or the result of a side effect of a user modification, such as a change in a data type. The class of a modification is determined as part of the semantic consistency checking process. During a user modification semantic action routines track the extent of the modification. Side effects of semantic changes are propagated and the affected program parts are also checked for semantic consistency. The semantic analyzer recognizes a change in the semantic consistency, which may affect the structural consistency. Thus, the semantic checking mechanism provides the context for structural consistency checking.

LOIPE does not attempt to place all modifications into the best possible class. The placement must be cost-effective, i.e., the resulting ability to continue execution must warrant the processing cost for both recognition of the class of a modification and the cost for the necessary corrections or adjustments to the program state. Thus, the desire to continue after a modification, such as modifications to issue debug actions, as well as the frequency of occurrence of such a modification taken into consideration in the placement of a modification.

3.5. Summary of the LOIPE Debugging Facility

LOIPE takes a novel approach to the debugging facility for a compiler-based language system. The debugging support is based on the program tree as the primary program representation. Both the program execution state and the debug state are integrated into the program tree representation. This allows LOIPE to provide a language-oriented, uniform interface for both program construction and program debugging. The user interacts with the system in a data-driven manner through editing operations. By making use of the incremental program construction mechanisms LOIPE is able to support truly interactive programming, i.e., program construction and debugging in arbitrary combination without loss in response time. Thus, the behavior exhibited by LOIPE at the user interface is very similar to that of interpretive systems, even though LOIPE uses compilation technology.

By operating on the program tree rather than the execution image the LOIPE debugger is

able to provide debugging activities that take advantage of the programming language and the abstractions provided in the program. One example is a powerful dynamic assertion checking facility. Debug functions that are commonly found in debuggers are supported by LOIPE. The following table gives a comparison of the compiler-based programming system Mesa, InterLisp as a prime example of interpretive systems, and LOIPE.

	Compiler-Based Systems Mesa	Interpretive Systems InterLisp	LOIPE
Method	Compilation	Interpretation and Compilation	Compilation
Supported Languages	highlevel strong typing	little typing	highlevel strong typing
Interactiveness	Editing and Debugging	Full Modification Cycle	Full Modification Cycle
Visibility of Program	Source Text Machine Code	Language-Oriented	Language-Oriented
Integration of Subsystems	Tool Set	Integrated System	Integrated System
Flexibility	Complete Programs Static Binding	Incomplete Programs Dynamic Binding	Incomplete Programs Static Binding
Mode of Interaction	Functional Multiple Command Sets	Functional Uniform Command Interface	Data-Driven Uniform Command Interface
Display-Oriented	yes	yes	yes
Expression Evaluation Conditional Debugging	Outside Language Subset of Language	Integrated In Language Support	Integrated In Language Support
Control Flow Monitoring	Explicit Debug Statement Single Step Code Patching	Anywhere Interpretive	Explicit Debug Statement At Any Grain Incremental Replacement
Variable Monitoring	At Procedure Entry/Exit	Full Monitoring Through Interpretation	Deterministic Monitoring Through Compiled Code
Continuation of Execution	After Debug Functions Only	After Modification To Active Procedures At Own Risk	After User Modifications Informs User If Not Possible
Optimizing Code Generators	Debugging At Machine Code	No Debugging For Compiled Code In Hybrid System	Limited Debugging Functionality

Figure 3-5: Comparison of Debugging Functionality

LOIPE's approach to providing debugging and monitoring support is a deterministic approach. The system supports monitoring of execution state at program locations that are defined a priori by the user. For example, modifications to data objects are monitored only in places where the user program is expected to change the value. Random write access to a data object due to some error in the code will be caught at the next expected modification location. Monitoring of data objects independent of the program logic requires checking after every statement in the program or interpretation of the execution image. With cooperation of the hardware or firmware, detection of random changes in the program state can be detected. Examples of such hardware support are machine instruction single-stepping on the PDP11 [PDP11 73], and exceptions on write access to specific memory locations, e.g., in object oriented systems [Jones 78]. We believe that deterministic debugging support is satisfactory, especially with the availability of relatively safe languages such as Mesa, Euclid, and Ada.

Chapter 4

Incremental Program Construction

During program construction the user goes through the steps *edit, compile, link, load, execute*, which we refer to as the *modification cycle*. In a traditional compiler-based system the user interactively edits the source program in text form. Then, the program is submitted to the compiler to detect syntactic or semantic errors and to generate object code if no errors are detected. Many programming systems support separate compilation. This permits only those parts of a program to be compiled that have been modified, reducing the time a user has to wait for an executable version of the program to be available.

Modern programming languages control the dependence between separately compiled program parts by supporting the concepts of localization of information and information hiding. Some programming systems, e.g., the Mesa system [Mitchell 79], use this dependence information to ensure that the program data base, i.e., both the source program and the executable representation are consistent. They cause all compilation units that are dependent on the modified unit to be rechecked by the semantic analyzer, and an executable representation to be constructed if no errors are encountered.

LOIPE takes a different view of the modification cycle. The time between editing operations can be utilized by LOIPE to process the modifications. The structure editor informs LOIPE of every editing operation through the action routine mechanism. Through these action routines LOIPE can do some processing of the modification cycle immediately, i.e., does not delay it until the user is ready to execute the program. In order to keep the processing cost between the interactions within limits, all steps in the modification cycle are applied frequently, but in a way such that only a limited part of the program is processed at any time. This is achieved by partitioning the modification cycle differently, and by reducing the dependence between program parts. As mentioned above, the module concept in modern programming languages

limits the dependence between program parts and permits these modules to be compiled separately. LOIPE extends this capability of incrementally processing program parts to the maintenance of the executable representation. It does so by reducing the dependence of program parts on physical information. Parts of the executable representation can be replaced without affecting other parts.

As Tichy points out in [Tichy 80], a program has logical properties concerning the semantics of the program, and physical specifications that are used in the generation of the executable representation. The semantic consistency depends only on logical, i.e., semantic information. Therefore, it can be checked separately from code generation. This property allows us to choose a different processing grain for semantic analysis and for code generation. The unit of processing for semantic analysis is chosen such that the user receives feedback on the source program when desired, i.e., while the user is still in context. Code generation is performed at a grain that is more suited to the incremental update of the executable representation.

The result is a modification cycle structure for LOIPE of the following form. Parsing of the source program is eliminated through the use of the structure editor. Semantic analysis is performed on the program tree in small steps as user modifications are progressing. Effects of the modifications on the executable representation are accumulated, and the executable representation is partially replaced independent of the grain of semantic checking. The executable representation is partially replaced by combining the code generation, and linking and loading step. Partial replacement can be done efficiently because the program part is accessible as program tree that is augmented with additional information and it is processed in context, i.e., all necessary information is available.

In LOIPE all steps of the modification cycle are performed sequentially. Alternatively, some of the processing steps could be performed concurrently with the editing activity. This approach has been tested in the SMILE system [Denny 81], where one piece of program text is compiled in a background process while the user edits the next program piece. We have refrained from taking this approach in LOIPE for the following reasons. All steps of the modification cycle not only access information in the program tree, but also modify some information. Elaborate mechanisms would be required to support sharing of the program tree between several processes and correctly synchronized update operations. The purpose of this effort is to guarantee fast response of interactions. As can be seen from measurements

on the prototype implementation of LOIPE, the use of the partial replacement mechanism permits sequential application of processing steps with sufficiently short processing time. Thus, we believe that the additional overhead for concurrent access to the program data base does not warrant the possible reduction in response time.

The remainder of this chapter consists of two sections. In section 4.1 we discuss LOIPE's support for incrementally maintaining a consistent program data base. This support includes repeated semantic analysis of program parts that are affected by modifications in order to determine the state of semantic correctness, and update of the executable representation in order to reflect the appropriate execution behavior. Section 4.2 elaborates on the provisions for incrementally maintaining the executable representation in accordance with the source program.

4.1. Incremental Consistency Checking

In LOIPE, semantic information on the program is already available in structured form, namely in the form of subtrees representing the definition sites of data types, object or procedures. This semantic information is directly updated by the user when he modifies the definition site subtree with the structure editor. LOIPE makes effective use of this semantic information by extending the name table mechanism provided by ALOE. Paragraph 4.1.1 discusses how the maintenance of semantic information can become an integral part of the program tree representation. The actual details of the resulting symbol table structure are not discussed here. The symbol table structure depends on the semantics of the specific language, i.e., scope rules, overloading, etc., and resembles that of compilers for the language.

The structure editor informs LOIPE of every operation on the program tree. Paragraph 4.1.2 discusses a mechanism in LOIPE that makes use of this information to perform semantic analysis at a small grain. This mechanism invokes semantic analysis routines of the form as they exist in compilers. Therefore, their details are not discussed here.

Incremental update of semantic information, as it is done in LOIPE, requires that program parts, whose semantic correctness depends on the modified information, are checked again. Paragraph 4.1.3 discusses the support mechanism in LOIPE that accomplishes this task.

Even though semantic checking and code generation can be performed separately at different grains, effects of modifications must be reflected in the executable representation. Paragraph 4.1.4 describes a mechanism that is used to communicate effects on the consistency of the executable representation, which are detected as part of semantic processing, to the partial replacement mechanism.

4.1.1. Availability of Semantic Information

ALOE enters identifiers in the program into a name table, and all references to an identifier are represented in the program tree as pointers to the same name table entry. This name table mechanism has been designed such that it can be extended to a symbol table mechanism through the action routine mechanism [Medina-Mora 81]. This allows us to implement a symbol table in the usual manner. Each name table entry has a list of symbol table entries linking the different defining occurrences of the identifier with the name contained in the name table entry. Each of these symbol table entries consists of a reference to the subtree representing the definition site. Once references to an identifier are bound to a specific symbol table entry rather than the name table entry, the semantic information about the symbol is directly accessible from any location in the program tree referring to that symbol. This is illustrated in Fig. 4-1.

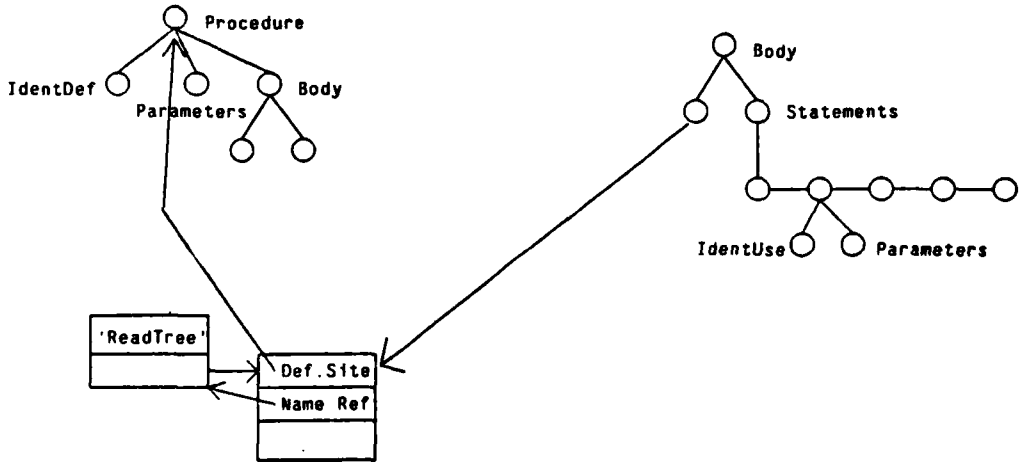


Figure 4-1: Definition Site Access Through Symbol Table

For example, type match of two operands of an assignment is determined by comparing the type nodes in the declaration subtrees of the two variables.

The symbol table entries are maintained by semantic action routines associated with the nodes representing identifiers. As in the formal semantic description of Ada [Honeywell 80] we distinguish defining and using occurrences of identifiers by two separate productions, *identdef* and *identuse* (see Appendix A.1). By doing so we avoid the semantic action routine to search the surrounding program tree in order to determine whether the identifier is a defining or using occurrence. The action routine associated with the *identdef* production maintains the symbol table, whereas the action routine associated with the *identuse* production attempts to bind the using occurrence to a legal definition.

The action routine for an *identdef* node adds a new symboltable entry with a reference to the definition site subtree when the node is created. The name table reference in the node is replaced by the symbol table reference. Similarly, when an identdef node is deleted the corresponding symbol table entry is removed.

Since modifications to a definition site directly reflect the change of semantic information, no updates are necessary in the symbol table entry. However, effects of the modification on the semantic information must be propagated to all use sites (see section 4.1.3).

Because semantic information is maintained incrementally rather than being regenerated, special care must be taken for defining occurrences of identifiers that conflict with already existing definitions. LOIPE could prevent the creation of defining occurrences of identifiers with conflicting names. ALOE's action routine mechanism allows the action routine for identdef to abort the creation when it detects the conflict [Medina-Mora 81]. We refrain from doing so because the enforcement of using only declared identifiers would limit the user's ability to modify the program freely. However, this requires that later removal of one of the conflicting definitions by a user modification does not accidentally delete the symboltable entry for the remaining definition.

4.1.2. Incremental Checking of Semantics

The structure editor informs LOIPE of every editing operation. Thus, it appears that semantics can be checked whenever a node is created. Its realization, however, presents some problems. For certain languages context information is necessary to identify the binding of a use site of an identifier to one definition site. Similarly, validation of operators in an expression requires availability of semantic information on the operands. During

construction of an expression information on operands is not available until both offspring subtrees are completed. Furthermore, the user can fill in components of an expression in arbitrary order. Thus, semantic processing of a node must be delayed until all context information is available. This means, that fully incremental semantic checking must propagate semantic state information along the program tree after every modification operation. Attribute grammars provide a notation to describe this information flow formally. Descriptions in form of attribute grammars have been used to automatically generate traversal patterns for semantic checking in compilers [Ganzinger 77]. Investigations of their applicability in the dynamic setting of an incremental, constructive environment are in their early stages [Teitelbaum 80, Kahn 81].

In LOIPE we opted for a different approach to incremental semantic checking. Instead of attempting semantic analysis after every modification operation, LOIPE invokes semantic checking on an enclosing program part, once all modifications in that program part are completed. The unit of the enclosing program part, which we refer to as *modification unit*, has been chosen such that it can be checked for semantic consistency without depending on adjacent program parts, i.e., only depending on the semantic context provided by defining occurrences of symbols. The statement was chosen as the smallest modification unit, because no semantic information is carried over from one statement to another. Nested statements are also checked separately. For example, the statement in the then-part of a conditional statement is checked independent of the condition. The condition is checked as part of the conditional statement.

Through action routines, LOIPE keeps track of the modification unit in which the modification is performed. A *modification context* is used to record the nesting of modification units which can be checked separately and whether a user modification has been performed. The modification context is realized as a stack whose entries consist of a reference to currently nested modification units that can be checked independently, and a dirty bit which applies to the top element on the stack. The dirty bit is set through semantic action routines whenever a user modification is made.

If the cursor leaves a modification unit either by exiting its subtree or by entering the subtree of a nested modification unit and the dirty bit is set, the subtree in question is submitted for semantic analysis. Semantic analysis on a modification unit is performed as it would in a compiler and is not elaborated here. Both entry and exit of the cursor in a

modification unit subtree are detected by the semantic action routine associated with the node representing the modification unit. Upon entry to a modification unit, the entered subtree is added to the modification context stack. The exit from a modification unit causes removal of the top element from the modification context stack, in addition to the checking for semantic consistency.

The result of the semantic analysis is recorded in the rootnode of the modification unit subtree. This state information is used by the error report and filtering mechanism to record its state information for repeated reporting (see section 2.3). This state information is also used to indicate to the code generator the state of executability. The code generator can then emit a call to the runtime system when such a unit is encountered.

4.1.3. Propagation of Side Effects

The validity of semantic consistency in a program part is dependent on the semantic context. Similarly, the consistency of the executable representation of a program part depends on the physical information that is supplied through the semantic context. Any change in the semantic context of a program part requires its revalidation of semantic consistency and potentially the replacement of its executable representation. A modification to the definition site of an item, i.e., that part of the definition subtree that contributes semantic information, potentially affects all program parts within the scope of the definition site, because it contributes to their semantic context. If the definition site is a type definition that is used in another type definition or an object declaration their semantic information is affected which must be propagated in turn.

LOIPE determines whether propagation is necessary by detecting through action routines whether the user modification is made in a subtree containing semantic information, e.g., the specification subtree of a procedure. Once the need for propagation is established, affected program parts must be located for reprocessing. Potentially all program parts in the scope of the modified definition are affected and would require checking. The cost of locating affected program parts can be greatly reduced by maintaining *use lists*, as has been shown by Mitchell [Mitchell 70]. Use lists record the binding of using occurrences of identifiers to a definition. By traversing the use list attached to a definition, all affected program parts are located. If for propagation complete use lists, i.e., use lists with every use site explicitly recorded, are traversed, the same replacement unit may be processed several times. This is the case if the

replacement unit contains more than one use site. This duplication of processing can be avoided at the cost of ordering use lists according to replacement units, or at the cost of propagating in two phases, first marking all affected units, and then processing them.

For LOIPE we chose a different solution, namely maintenance of a *replacement unit use list*. For every use site of an identifier, the enclosing replacement unit is added to the list, if it is not yet included. The cost is proportional to the length of the replacement unit use list, which is considerably shorter than a complete use list. Use sites are only added to the use list when being bound.

During propagation of the replacement unit use lists, each affected replacement unit is encountered only once for processing, independent of the number of use sites. As a replacement unit is processed the actual use sites that are involved in the propagation are detected by the semantic analyzer. It checks for every use site of an identifier whether it is bound to the definition site causing the propagation. This information is useful for the error reporting mechanism to avoid reporting errors during propagation that are not connected to the propagation.

The propagation mechanism must be able to handle addition of semantic information, and removal of semantic information correctly. Semantic information is added by the creation of a defining occurrence of an identifier. The new definition may cause previously undefined use sites to become defined. However, the use sites are not contained in any of the replacement unit use lists of the symbol table entries for the given identifier. In order to be able to locate the undefined use sites other than by complete search of the program tree, LOIPE maintains a list of replacement units containing the undefined use sites with the name table entry. Addition of semantic information may also require previously bound using occurrences of identifiers to be rebound. Such use sites are located by finding the symbol table entry of a definition whose scope is overwritten by the new definition and propagating with its use list.

The deletion of a declaration implies the removal of semantic information, causing using occurrences previously bound to that definition site to become invalidated. Semantic action routines inform LOIPE of a deletion of a node before the actual removal. This allows LOIPE to utilize semantic information, e.g., replacement unit lists, to propagate the efforts of the removal and update the binding of use sites. However, since the node has not actually been removed yet, the semantic analysis process must make sure that the definition site, which is

being deleted, is used when processing the replacement units that are located by the propagation mechanism.

4.1.4. Consistency of the Executable Representation

The consistency of the executable representation is affected by:

- actual modifications to the user program,
- change of the state of executability as a result of changes in the semantic information, and
- changes in the physical information.

LOIPE uses the partial replacement mechanism to update pieces of the execution image as they become affected. Even though the grain size for partial replacement can be chosen independently of the grain used for semantic checking, the results of the semantic checker must be communicated to the partial replacement mechanism to inform it of program pieces whose executable representation is invalidated. For that purpose LOIPE maintains a *replacement context* much in the same way as the modification context. The replacement context consists of a stack of nested replacement units and an *invalidation indicator*. The stack is managed by action routines for the replacement units. The invalidation indicator records whether the top element of the replacement context must be updated in the executable representation. If the cursor leaves a replacement unit (either by entering an enclosed replacement unit or by returning to the enclosing replacement unit) and the invalidation indicator is set, the program part is submitted for processing to the partial program replacement mechanism. For a discussion of the choice of the replacement unit we refer to section 4.2.

It is the responsibility of the incremental semantic checker to set the invalidation indicator. The semantic checker does so whenever it encounters a dirtied modification unit, a change in the state of executability of a modification unit, or a change in the physical information. The latter two cases can occur without an explicit user modification to the given program piece due to side effects of other user modifications. The propagation mechanism discussed in paragraph 4.1.3 broadcasts these side effects by reprocessing the affected program pieces.

4.1.5. Summary of Incremental Consistency Checking

Semantic checking and consistent maintenance of the program data base is an integral part of the program construction support in LOIPE. Semantic information is maintained as part of the program tree. Semantic correctness is checked in short intervals to give the user feedback while still in context. Effects of changes to semantic information are propagated immediately in order to keep state information in the program data base uptodate. Similarly, consistency between the source program and its executable representation is maintained incrementally. Execution can be attempted at any time without casuing long delays, and the execution will show the behavior expected from the source program view.

4.2. Partial Program Replacement

This section deals with the mechanisms for generation and maintenance of an executable program representation that is consistent with its source representation. In compiler-based environments, program execution is performed with a statically preprocessed representation of the program. All instructions from the source program are directly mapped into instructions of the computer, and all interconnections between program parts, i.e., references, are bound before execution. The result is efficient execution of the program, static binding of interconnections between program parts, however, also results in a strong dependence between pieces of the executable representation. This dependency makes maintenance of the executable representation difficult undertaking. In traditional compiler-based systems, the maintenance issue has been avoided by requiring reconstruction of the complete executable image to incorporate a change. A partial update of the executable representation would have extensive side effects due to the network of reference dependencies.

The dynamic linking facility of Multics [Organik 72] is an early example of runtime support that permits addition of program parts without a complete reconstruction or high cost of replacement. This is achieved by delaying some of the binding process to execution time, thus reducing the interconnection dependencies.

For large long-running software systems, dynamic upgrading of software parts under realtime constraints, such as airline reservation systems or telephone switching systems, is essential . As Fabry points out in [Fabry 76], the introduction of a level of indirection provides

the flexibility necessary to replace program pieces on the fly. Even the incremental update of data structure modifications in a running system has been investigated [Goulon 78]. In these approaches, all invocations of the old version of a program piece complete under the old version. In LOIPE this is not acceptable because program modifications are expected to be reflected in the executing program immediately. This is crucial during program debugging, where the actual program behavior must correspond to the behavior, expected in the source program. However, the idea of a level of indirection can be used to realize the effect of an immediate partial replacement in the executable representation.

The next section elaborates on the use and cost of indirection to permit incremental update of the executable representation. In section 4.2.2 we discuss the effects of the resulting virtual machine for the executable representation affects the three processing steps in partial replacement, i.e., code generation, binding and loading. Finally, section 4.2.3 discusses the feasibility of interactive, remote program development and execution.

4.2.1. Use of Indirection

Linking is the process of binding elements in the symbolic name space to elements in the physical name space. The *symbolic name space* consists of names of items in the source program representation, e.g., procedures, data objects, whereas the *physical name space* is made out of addresses to locations in the execution image, i.e., the address space of the user program executing on the target machine. Static binding maps symbolic names to physical addresses during the construction of the execution image. This mapping can only be changed by rebinding.

4.2.1.1. Indirect References

Indirect references introduce an intermediate name space, which we refer to as *placeholder name space*. This third name space allows for a two-step mapping. Symbolic names are statically bound to elements in the placeholder name space. Each element of the placeholder name space defines a mapping to a physical name for an item. This mapping, however, is performed dynamically by the hardware using indirect addressing mode. Because the second mapping is performed dynamically, physical names can be changed, items can be moved, without affecting the static binding of symbolic names to placeholder names. A component within an item is referred to by the pair (

pair is known statically. The address of the component can be generated using an index-deferred addressing mode. Since components within an item are bound statically relative to the item, they cannot be rearranged without affecting the bound name. Items themselves can be moved around, necessitating only the update of the dynamic mapping information in the appropriate placeholder.

The placeholder name space is realized as a reserved set of storage locations in the address space of the executing user program. Each of these locations contains the physical name of the item being referred to. The cost for the use of the intermediate name space is the storage space for the placeholders, and an extra memory reference at runtime when the dynamic mapping is performed.

The executable representation of a program consists of code and data. Flexibility of the executable representation is achieved by placing both code and data items into the physical name space and assigning placeholders for them. Resemblance of this indirection mechanism with capability systems is not accidental. In both cases an appropriate choice must be made for the grain size of units referred to by placeholders, such that the gained flexibility justifies the cost incurred by the placeholder mechanism. In the following two paragraphs we discuss the choice of replacement unit for code, and argue for not using indirection for all data objects.

4.2.1.2. Unit of Partial Code Replacement

Several considerations must be taken into account when chosing the unit of partial code replacement.

- All references between replacement units must be bound to placeholder names in order not to be affected by a replacement, which may store the new code piece in a different physical location.

- The size of the replacement unit determines the cost of partial replacement, i.e., code generation, binding, and loading. Since partial replacement is performed between user interactions, the processing time must be kept beolow a certain threshold in order to maintain fast response time. Acceptable The response time should not be more than a couple of seconds.

- Code generation and optimization require a certain context. If the replacement unit is smaller than the context, context information must be updated incrementally.

Based on these considerations, the procedure has been chosen as the replacement unit. The procedure defines a program part with exactly one external reference point, the entry to the procedure. All references to locations within the procedure are confined to the procedure body. They can be resolved as part of the partial replacement process (see section 4.2.2). The entry of a procedure is only referenced in the context of a procedure or function invocation. Thus, the access via placeholder can be performed by an indirect call instruction. Indirect procedure call is familiar in implementations of compiler-based languages. It is supported by several language systems, e.g., Mesa [Mitchell 79] or Perq Pascal [Barel 81]. The procedure provides a satisfactory context for most optimizations with the exception of inter-procedure optimization.

The processing time for partial replacement is kept to a minimum in LOIPE. Code is generated from a semantically correct program tree representation. As will be discussed in section 4.2.2, the three steps of partial replacement can be closely integrated, because the program part is processed in context. The result is efficient processing during partial replacement. The average size of procedures tends to remain within one page of source program. Measurements on a prototype implementation of LOIPE have shown that for procedures of less than one page replacement takes about two seconds (see section 6.1.2).

4.2.1.3. Incomplete Indirection for Data Objects

The use of a placeholder for each data object results in a proliferation of placeholders. Therefore, we investigate in this paragraph whether indirection for all data objects is necessary to support partial replacement. We argue that indirection through placeholders must not necessarily be provided in LOIPE.

Indirection is inherent to the implementation of some types of data objects, even though the user may not be aware of it. Arrays with dynamically determined dimensions or flexible arrays, i.e., arrays whose dimension changes during their lifetime, are implemented through dope vectors. The access to an array element is performed through the dope vector. Thus, dope vectors can be considered placeholders for arrays.

Local data objects are usually stored in procedure activation records and are accessed relative to its base address. This indirect access permits references to these data objects to be bound independent of the physical location of the activation record. However, addition, modification, or removal of a local declaration affects the binding of these relative references

and the layout of the activation record. The relative references are rebound when the procedure is recompiled. Activation records whose layout has changed are removed through *unwinding* (see section [execcontrol]).

Rebinding of direct references to global data objects is required when the physical position of data objects changes. A change of the physical name space position is caused by an increase in the size of the data object beyond the space allocated for that data object. A reduction in size of the data object does not require repositioning because the old space can be reused. The size of a data object changes, either if the data object is an array and the array bounds are extended, or if the data type of the object is replaced or modified. Such modifications, however, frequently also have effects on the semantic or physical information at the usage site of the data object, which must be propagated. For example, the code of an assignment statement includes the number of bytes to be moved. Similarly, information on the range of an array may be recorded as part of a loop accessing the array. Propagation causes reprocessing of the affected program parts, i.e., parts containing use sites. As part of the reprocessing, the use sites are bound to the new location.

Updating of references to dynamically created data objects is not an issue, because modification of the type of a data object is a fatal modification requiring reinitialization of the program execution state. Therefore, an implementation of data objects without explicit placeholders is satisfactory for LOIPE.

4.2.2. Cooperation of Processing Steps

In LOIPE all steps of the modification cycle are closely coupled through the program tree. This allows us to design these processing steps to be cooperative. One example of cooperation is the provision of support for code generation by the semantic checker. Physical information is made available in the program tree. Furthermore, during propagation of side effects code can be generated as the semantic checker traverses the procedure subtree.

The code generator also cooperates with the linker. It performs the linker's task of name binding. Code is produced for semantically correct program parts. By assigning an entry in the placeholder name space for a procedure as soon as the specification, i.e., the procedure name, has been created, the placeholder name of the procedure is available for binding even before the procedure itself is complete or semantically correct. Similarly, from a complete

data object specification the size of the data object can be derived and space assigned. Thus, binding information for global references is available to the code generator. Local references can also be bound by the code generator because both definition and use sites reside within the procedure. However, this requires two passes of the code generator, because forward references to code locations may occur, e.g., in the code for a conditional statement. The second pass can be considered the binding step for local references.

The code generator can produce location independent code. All global code references must be expressed in absolute terms, and local references must be represented by relative references. For local references this means activation record relative addresses to local data objects, and program counter or procedure entry relative addresses to local code locations. Location independent code eliminates relocation of code references. Thus, once the size of a code piece being generated is determined at the end of the first pass of the code generator, the emitted code can be placed directly into an appropriate location in the physical name space without buffering the complete code sequence in the second pass. Code generation, binding, and loading are combined into two traversals of the procedure representation.

Cooperation between system parts is necessary for the management of the physical name space, i.e., the address space of the process executing the user program. The partial replacement mechanism of LOIPE incrementally requests assignment and deassignment of space in the physical name space over the lifetime of the executable program representation. If the supported programming language provides dynamic creation of data objects, user programs may also request assignment and deassignment of physical name space during execution of the program. In this case a common space manager must be used to handle the request. An alternative solution, separate management of two disjoint partitions of the physical name space has not been considered, because the estimation of a reasonable partition is difficult.

4.2.3. Remote Program Execution

Compiled programs can be executed by themselves with limited runtime support. The source program does not have to be present. This permits standalone execution of programs in a separate process or on a different machine without program development support. Runtime support for interactive program debugging can be provided either by linking debug software into the user program, or through a separate debug process. The second approach

is preferred over the first for several reasons. The user process with its own address space protects the development and debugging system from damage by a faulty user program. The user does not have to decide until runtime whether debug support is desired. By executing separately from the user program, the debugger cannot be preempted from communication with the user by the user program. This is an important consideration if the program consists of several processes, e.g., for Ada *tasks*. The user is able to multiplex the debugger for monitoring of each process in one environment [Swinehart 74]. This approach, however, requires that the debug process has access to the address space of the user process, and can control its execution.

In LOIPE the user program is executed remotely in a separate process. Access to the user process, whether on the host machine or a different machine, is provided through one module, the *remote access* module. The specification of the module is chosen such that LOIPE is independent of the specific mechanism that is used to access the user process. LOIPE's implementation of this module is based on a message scheme. The routing of communication is handled by the message system. The user process contains runtime support that cooperates with the LOIPE process. Operating system support, e.g., the Unix *ptrace* function, is required if the user process cannot be expected to cooperate. The remainder of this paragraph discusses the specification of the remote access module.

The interface of the remote access module is defined such that it allows for efficient implementation for different situations. This is evident in the set of functions provided for updating the execution image. Both code and data are updated through block write operations. The information to be written is either produced by the code generator, or is available on permanent storage. In the first case the information is placed into the user process as it being generated with the operations *WriteBlock* and *AppendBlock*. WriteBlock starts writing a block of information, indicating the total amount to be stored. Repeated calls to AppendBlock supply the information to be written. The effect is pipelining of the information from the code generator through the remote access module into the user process. In the second case the remote access module is passed a reference to the permanent storage as part of a call on the operation *LoadBlock*. This gives the implementor of the remote access module several alternatives for retrieval of the information from the file and storage into the user process.

The program construction facility must aquire space in the user process to store new

program parts. When program parts are replaced both new space may have to be allocated and used space may become available. For that purpose the remote access module provides two operations, *GetUserSpace* and *FreeUserSpace*. Knowledge as to whether a space allocation mechanism is available as part of the runtime system in the user process, or the user process address space is managed by a simple mechanism in the remote access module is confined to the remote access module.

The remote access module also provides operations necessary for program execution and program debugging. Program execution is initiated through one of two operations, *StartExecution* and *ContinueExecution*. StartExecution causes execution to begin at the specified location, remembering the previous program execution state. ContinueExecution causes execution to be resumed at the location indicated by the current program execution state. Both operations are asynchronuous, i.e., execution of the user process proceeds in parallel with the execution of LOIPE. LOIPE waits for the result of an execution through the operation *WaitOnExec*. This operation returns upon a report from the user process. The user process informs LOIPE of normal termination of execution or of exceptions that have been encountered. Exceptions include hardware exceptions such as divide by zero and software exceptions such as trace or break points. Exceptions leave the user process suspended at the point that caused the exception. Exceptions are parameterized, i.e., the program location raising the exception can be reported in terms of program tree references for predetermined locations (break and trace points). An asynchronous form of the exception mechanism, that does not suspend the user process, is available as an event generation mechanism for monitoring purposes.

The remote access module also includes operations to read from the user process address space. The read operations are used to retrieve the program execution state, i.e., the content of data objects and the runtime stack. The amount of information to be transferred is limited, because LOIPE issues read requests only for those parts of the program execution state actually being displayed to the user. One read operation, *ReadBlock*, retrieves a specified block of data from the user process. Another read operation, *ReadCallStack*, extracts the list of procedure activations from the runtime stack. The list consists of pairs of references in the physical name space to the active procedure and its call site. By providing this special operation reading of the whole runtime stack is avoided. Finally, the *ReadFrame* operation allows LOIPE to read data from one activation record by referring to it as frame number, i.e., position number of the activation record on the callstack, and offset. Both local variables and

parameters can be retrieved with this operation. In addition to the read operations, the remote access module supports program debugging through a *WriteFrame* operation and operations for updating the callstack structure in the runtime stack. The latter set consists of an operation for unwinding the set of active procedures, an operation for updating current execution points, and an operation for marking current execution points, i.e., return addresses, as nonexecutable. A current execution point is marked nonexecutable by replacing its physical reference on the runtime stack with a reference to an exception handling routine in the runtime system that reports the exception back to the remote access module.

4.3. Summary of Incremental Program Construction

Incremental program construction is critical to the success of LOIPE as an interactive, compiler-based programming environment. In this chapter we have demonstrated how incremental application of all processing steps in the modification cycle is used consistently. The result is maintenance of a program whose executable representation is consistently brought up to date between user interactions such that the program can be executed at any time. Incremental processing means that existing information is updated rather than regenerated. For this purpose an executable representation has been chosen that permits its maintenance with limited side effects. This tradeoff between maintenance of information and its regeneration occurs repeatedly in LOIPE. The different parts of LOIPE perform their task in context, i.e., in a well-defined environment. Therefore, they can be tailored to each other and can share information, resulting in a simple system structure with limited redundancy of information and support mechanisms.

The program tree acts as a central depository for information, which is maintained by the structure editor in cooperation with other system parts through the semantic action mechanism. Some semantic information is available as an inherent part of the program tree representation, whereas other information is derived from the source program. Since derived information can be regenerated, permanent maintenance is not necessary. However, it must be regenerated quickly when the demand arises, because all processing is done between user interactions. The alternative, maintenance of information also has a certain processing cost. LOIPE uses a combination of maintenance and regeneration of information as can be seen in the realization of incremental consistency checking. By doing so and working with the program tree, which can be easily accessed and manipulated, LOIPE is able to exercise the

modification cycle frequently between user interactions without destroying the interactiveness of the system behavior. Measurements on a prototype implementation support this claim (see section 6.1.2).

Chapter 5

Realization of Language-Oriented Debugging

In LOIPE the source program is synonymous with the program tree. Both program construction and debugging are handled in the same way by manipulating the program tree through a structure editor. An obvious realization of program execution and interactive debugging is the *interpretation* of the program tree. Such an implementation of debugging support violates one of LOIPE's premises, the compilation of programs. Therefore, we have to consider debugger implementations for compiled code. Section 5.1 discusses several alternatives including the approach taken in the LOIPE system.

In a compiler-based system the source program is translated into *object code*. This program representation is executed directly on the hardware. During execution the hardware maintains the current program state in terms of the object code on a *runtime stack*. In LOIPE the program state is presented to the user in the form of the program tree representation. Therefore, LOIPE must map snapshots of the actual program state into the program tree at appropriate times. Section 5.2 elaborates on the necessary support mechanisms for this mapping.

The translation of a source program into its executable representation, the object code, is not unique. The code generator can improve the quality of the object code through the use of optimization techniques. In order to optimize a program, the code generator treats the program as a single static entity. This means that the code generator may change the order of evaluation for some constructs without affecting the overall program behavior. The treatment of the program as a single static entity conflicts 1) with the debugger's ability to examine the executing program at any point, and 2) with LOIPE's notion of incremental modification of the program. Section 5.3 discusses this problem and proposes an approach that permits LOIPE to support a certain degree of code optimization.

5.1. Realization of Debug Actions

A basic function of the debugger is the insertion and enabling of debug statements in a program. These debug statements are entered into the source program, in the case of LOIPE, the program tree. Since the user may not know a priori what debug actions should be taken, he must be able to interactively enter and enable them. When executed, enabled debug statements cause the debugger to report the progress of execution to the user.

Interpreters lend themselves well to the support of interactive debugging. The interpreter works with an intermediate representation that closely resembles the program text, for example, the program tree. In the context of LOIPE that means that the source program and the executable representation are identical, i.e., the mapping of debug statements and the program execution state is trivial. Due to this fact and the delayed binding of references, the interpreter is flexible enough to adjust program execution immediately to program changes.

In contrast to interpreters, compiler-based systems maintain a second program representation, the object code. Object code is executed directly on the hardware. It is executed efficiently because the code generator does as much static processing as possible, e.g., references are bound at compile time. Such an executable representation has the disadvantage of being rather inflexible for modifications. Existing compiler-based systems have embedded program modifications into the object code by compilation and complete relinking of the execution image. Complete relinking destroys the program execution state. It can, therefore, not be used to implement interactive insertion and enabling of debug statements.

Existing compiler-based debugging systems have taken two approaches to overcome the problem of losing the program execution state when inserting debug statements. In the first approach code for debug statements is inserted into the object code at compile time. The debugging system provides runtime support that allows the user to selectively enable the inserted debug statements during program execution. This approach has two drawbacks. First, all debug statements that can be enabled interactively must be defined at compile time. Second, debug statements contribute overhead to the size of object code and the execution time even though they may not be enabled.

In the second approach, known as *code patching*, the debugger inserts code for a debug

statement by replacing instructions in the object code. The insertion is made interactively, avoiding the runtime overhead for unused debug statements. The instructions are patched in such a way that the program execution state is not damaged and execution can be resumed. The drawback of this process is that the debugger requires detailed knowledge of the object code, i.e., the machine architecture and the way the source program is translated.

LOIPE's debugging facility is not based on the object code, but on the program tree. It has no knowledge of the object code. All changes to the program tree, including enabling of debug statements, are inserted in the object code through a single mechanism that is provided by the incremental program construction facility, i.e., *partial program replacement*. This mechanism updates the execution image by replacing pieces of object code without requiring complete relinking. However, this replacement may have side effects on the program execution state that must be corrected in order to permit resumption of execution. This approach of implementing debug statements is being compared to the other approaches that have been outlined above.

5.1.1. Interpretation of the Program Tree

Although we precluded the use of interpretation for LOIPE, a closer look at interpretation reveals some features that are valuable for debugging support. Interpretation of the program tree permits implementation of language-oriented debugging facilities in a simple manner. Debug statements are defined by inserting them into the program tree, and enabled by tagging the appropriate construct. As pointed out before, the program tree is the executable representation, and source program modifications are immediately available for execution. During execution the interpreter checks for the tags and suspends execution if necessary. Single stepping and tracing of language constructs throughout the program is also easily implemented. The program tree not only is the source program representation, but also contains semantic information such as a description of the interdependence of program parts. This information is available as part of the execution context and can be utilized at runtime.

The interpreter advances the control flow of a program and performs all accesses to data objects. Consequently, the provision of variable monitoring is trivial. The interpreter can also keep a record of all changes to the program state during execution, be it control flow or data flow. Such a record of program state changes nicely supports the realization of a reverse execution facility, such as InterLisp's *undo* facility [Teitelman 78].

Because interactive debugging support at the language level is provided with little additional effort, interpreters have become popular as the basis for language-oriented programming environments [Teitelman 78, Archer 81, Shapiro 80, Teitelbaum 80]. In existing interpreters debug statements must be defined at the location of their evaluation (with the exception of single stepping). Debug statements with larger application scopes, e.g., assertions for a module, require additional support to locate the actual evaluation sites.

The flexibility of interpretive systems to reflect user modifications in the program execution immediately is achieved at the expense of execution time. Some hybrid systems, e.g., Interlisp, have tried to overcome the high cost of interpretation by supporting code generation of program parts. Any use of debugging facilities forces the interpretation of the program parts being debugged. The hybrid system must be able to switch between execution of compiled code and interpretation of the program tree. Furthermore, both modes of execution must produce the same program behavior, a difficult task if the program is sensitive to timing. During execution the interpreter must be resident on the machine executing the program even though all program parts may be compiled because they are linked together by interpretation. This means that the interpreter's runtime system, a substantial amount of code, must be available on the machine executing the program.

By assuming the LOIPE system to be based on compilation only, we give up some of the features inherent to interpretive systems. First, a second representation, the object code, must be consistent with the program tree. Second, the object code, which is the executable representation, is generated by statically binding program parts. The static binding makes the object code inflexible to modifications. This inflexibility must be overcome in order to support interactive program changes and debug actions. Third, semantic information that is stored with the program tree is used for code generation, but not passed on to the object code. It is, therefore, not accessible to the debugger's runtime support in the execution image. This must be compensated for by LOIPE explicitly supplying debugging code for all locations, at which a debug statement must be evaluated.

The execution of object code on the hardware is a form of interpretation. But the hardware interpreter does not provide any support for recording changes to the program state during execution and for monitoring access to selected memory locations. Therefore, the necessary support for variable monitoring and reverse execution must be implemented in software. Code must be inserted explicitly at all appropriate locations in the execution image. We

recognize this as a deficiency of the compiler-based debugging approach as long as no additional support is provided by the hardware interpreter.

5.1.2. Patching of Object Code

Code patching exists in two forms. In the simple form signalling instructions are inserted in the object code by overwriting existing instructions. The signal activates the debugger. The debugger then locates the debug statement that is associated with the signal and evaluates it interpretively. When execution resumes the overwritten instruction must be executed. This is commonly done by using the single instruction trap mechanism. This simple form of code patching has the disadvantage that an interpreter for a large part of the supported language must be available, because the interpreted debug statements can be quite elaborate.

In the other form of code patching debug statements are translated into object code. This object code is threaded into the existing execution image by rerouting the control flow to the inserted code with a patch. [Deutsch 71] discusses the use of this mechanism for interactive insertion of performance monitoring probes.

Both forms of code patching require write access to object code. This is in conflict with the common practice of placing code into the read-only section of the execution image. This problem is usually resolved in one of two ways. Either the write protection for the code area of the execution image is lifted, allowing even the user program to modify code, or the debugger process is given special access privilege to the execution image. One example is the *ptrace* facility of Unix [Unix 81a] that gives a parent process (the debugger) modification rights to the address space of any of its offsprings (the user program).

The code patching mechanism must have detailed knowledge of the machine architecture and the runtime system of the supported language in order to perform the patch without doing damage to the execution image or to the program execution state. This means that this knowledge must be embedded into both the compiler and the debugger. Any inconsistency between the execution image and the runtime stack would prevent the resumption of execution after enabling or disabling of a debug statement. There are cases where resumption of execution cannot be guaranteed after patching of debug statements. One case is the occurrence of a runtime error in the code of a debug statement, e.g., in an assertion. The correction or removal of the debug statement cannot be easily fixed for execution

resumption because the error occurs at a random program location. The second case concerns the use of debug variables. Debug variables, whose scope is local to a procedure, are allocated with the local variables. Their interactive insertion changes the activation record if the procedure is active at the time of insertion.

LOIPE encourages frequent alternation between program construction and debugging. It must be able to keep the debug context, i.e., the set of enabled debug statements, invariant to program modifications. Traditional compiler-based systems include a program modification into the object code by compiling the modified source part and complete relinking. The newly generated execution image must be repatched with all enabled debug statements in order to restore the debug context for execution. This is a costly undertaking, especially if the modified program part did not contain any enabled debug statements. Complete relinking of the execution image destroys the program execution state on the runtime stack. The program execution state is lost even if the modified program part is not located in one of the procedures on the runtime stack. By extending the code patching mechanism to patching of program modifications, the execution state could be preserved in those cases, where the structural consistency of the source program and the program state are not invalidated (see section 3.4.2). Code patching of all modifications, however, results in a increasingly fragmented execution image, and correct patching without damage to the program execution state becomes a problem. Thus, the code patching approach deals with the same issues concerning resumption of execution as partial replacement, but must support two mechanisms (compilation/linking and code patching) instead of one.

5.1.3. Debugging Through Partial Replacement

As we showed in chapter 4, partial replacement overcomes the inflexibility of object code through the use of indirect procedure references. A change of the source program is included in the object code by generating code for the enclosing procedure and its replacement in the execution image. LOIPE uses this mechanism to support incremental program construction and to implement interactive debugging. This reduces the complexity of the system because the executable representation is maintained by a single mechanism. Furthermore, all modifications to the source program including enabling of debug statements are treated uniformly. For every modification LOIPE decides whether the resulting update of the object code has side effects on the program execution state and whether the effects can be corrected. LOIPE must be able to keep the program execution state consistent with the

updated object code for a class of modifications that includes enabling and disabling of debug statements. Otherwise, execution could not be resumed after a debug interaction. Before we elaborate on the support for resumption of execution, let us examine the cost of implementing debug statements by partial replacement.

Setting and enabling of a single breakpoint requires the enclosing procedures to be recompiled and relinked. The cost of setting a breakpoint by partial replacement is higher than the cost of patching it, but the processing cost is still affordable for interactive debugging, as can be seen from the measurements on the prototype (see section 6.1.2). Partial replacement fairs even better in relation to code patching when the enabled debug statement has a larger application scope, i.e., must be inserted several times, or when several debug statements are enabled. For example, on enabling single stepping of statements in a procedure all necessary debugging code is inserted in one processing cycle of the partial replacement mechanism. Code patching would require a separate patch after every statement.

For enabling single stepping of statements, partial replacement cannot compete with interpretation, especially if the whole program is to be single stepped. However, LOIPE's high level debugging facilities, such as dynamic assertion checking at various abstraction levels, should eliminate the need for single stepping or reduce it to single stepping in individual procedures.

In section 3.4.2 we discussed the effects of different modifications on the program tree representation of the execution state. The replacement of a procedure due to a modification in the object code may also have an effect on the runtime stack if the replaced procedure is active. The runtime stack consists of a sequence of procedure activation records, each representing an active procedure invocation. An activation record contains 1) a reference to the callsite, 2) the actual parameters and local variables of the respective invocation, and 3) additional state information that is specific to the implementation of procedures on the given hardware, e.g., use of general purpose registers. The additional state information changes after a modification has been made to a procedure. For the moment we limit the discussion to non-optimizing code generators and refer the reader to section 5.3 for a treatment of optimizations. The layout of parameters and local variables changes if either of them has been modified in the source program. In that case we have already determined in section 3.4.2 that the affected activation record must be removed by unwinding the runtime stack in

order to permit continuation of execution. This leaves us with the correction of side effects for references to the callsites.

References to callsites, also known as *return addresses*, are direct pointers into the execution image. They can be damaged for two reasons when the procedure containing a callsite is replaced: 1) the object code for the procedure must be relocated because it does not fit into the space provided for the previous copy of the code; 2) the position of the callsite relative to the base of the procedure object code changes because the procedure source code in front of the callsite has been modified. In both cases it is possible to correct the callsite reference on the runtime stack. From section 3.4.2.2 we know that callsite references in terms of the program tree are invariant to user modifications (with the exception of removal of an active callsite which causes unwinding of the runtime stack). We also know that LOIPE updates the program tree representation of the callstack whenever program execution is suspended. Thus, an invariant form of callsite references is available when modifications are made. After a modification, the new execution image location of any active callsites are determined by the code generator and can be updated in the runtime stack. The mapping of references between the two program representations is discussed in section 5.2.1.

5.1.4. Summary

The realization of debugging support through partial program replacement is feasible. Measurements on a prototype confirm that the cost of translating a whole procedure for insertion of debug statements is acceptable. Partial replacement is chosen over code patching because it treats modification to the source program and enabling of debug statements in the same manner, reducing the complexity of LOIPE. We recognize that the use of an interpreter has advantages over the execution of compiled code. Control flow tracing for the whole program at the statement level is trivial for an interpreter, whereas its provision in the context of partial replacement is costly. However, LOIPE provides high level debugging facilities that reduce the need for single stepping. Without additional support from the hardware "interpreter" for reverse execution, LOIPE resorts to unwinding of the callstack and to recovery techniques for restoration of program state.

5.2. Accessibility of Program State

The program state is visible to the user in the form of the program tree representation, whereas the executing hardware changes the program state in the execution image. These changes must be reflected in the program tree in order for the user to have a consistent view of the progress of execution. The program state consists of the callstack, i.e., a list of all active procedures and their callsites, and of the current contents of global objects and local objects of active procedures. This can be a considerable amount of information. LOIPE only maps those parts of the program state into the program tree representation that are requested to be displayed. The callstack is always updated in the program tree when execution is temporarily suspended. The current value of data objects is only mapped if the data object is being monitored in the monitoring window, or the user has issued an explicit display request (see also section 3.2.2.2).

We continue by elaborating on the support mechanisms for the implementation of the mapping between the two representations. Section 5.2.1 discusses the provision of information for mapping the control flow and its use for correction of side effects in the runtime stack. In section 5.2.2 we present an access mechanism to data objects in the execution image that allows display and modification.

5.2.1. Mapping of Control Flow

The relationship between locations in the source program and the object code is established by the compiler and the linker/loader. The compiler assigns object code locations relative to compilation units, and the linking/loading step determines absolute positions in the execution image. This information must be made available to the debugger to locate source program positions for execution image references and vice versa.

In existing systems the compiler passes this mapping information on to the debugger in a *mapping table* that is produced together with the object code. This mapping table contains reference pairs to source code and object code for all program locations that are expected to be mapped. These locations include those that are marked by a user defined symbol, e.g., a procedure name, and predefined locations in the source program. For source program debuggers such predefined locations are usually lines of program text. The debugger searches the mapping table in order to locate a source program reference that corresponds to an object code reference.

The cost of maintaining the mapping information and looking up a mapping is high because all program locations that could potentially be mapped are included in the table. In the case of the source-level debugger sdb on Unix, the size of the mapping table can be a factor of ten larger than the object code. In the case of LOIPE the size of the mapping table would increase further because the program tree allows for a finer grain of program locations than lines of program text. Therefore, we consider other means of providing the mapping information. First, paragraph 5.2.1.1 analyses the set of program locations that must be mapped between the two representations. Then, different ways of actually keeping mapping information and making it available when needed are discussed in paragraph 5.2.1.2. Finally, paragraph 5.2.1.3 presents a hybrid solution that trades off between different representations of mapping information for different types of program locations.

5.2.1.1. Mapping of Program Locations

Debugging systems must be able to map program locations in both directions. The mapping from the source program to the object code is necessary for modification or correction of the execution image. Debuggers that implement debug statements through code patching are given a location in the source program at which the debug statement should be inserted. This reference is mapped into the object code location that is to be patched. For that purpose, mapping information must be provided for every program location at which a debug statement could potentially be inserted. In the case of source program debuggers this is every line of source program text. LOIPE implements debug statements through partial replacement. The mapping of debug statements and their location into object code is taken care of by the code generator. Therefore, no explicit mapping information must be kept for potential application sites of debug statements.

The replacement of a procedure in the object code may require correction of side effects. First, the incremental loader must be able to locate the placeholder of the replaced procedure to update its reference to the actual location of the procedure's object code. Second, replacement of a procedure requires correction of callsite references, if the procedure is active. Any callsite reference to the active procedure on the runtime stack must be adjusted to refer to the appropriate location in the newly generated object code. As pointed out in section 5.1.3 such references are corrected by deriving the new object code location from their program tree location.

The debugger maps control flow information in the program execution state, as it is

recorded on the runtime stack, into the program tree in order to visualize it for the user. This control flow information consists of the list of active procedures with their current execution point. As explained in section 3.2.1 the current execution point of an active procedure points to the callsite of another active procedure, to a debug statement that caused execution to be suspended, or to the location of a runtime error. The current execution point is recorded on the runtime stack as the return address in the next activation record. The return address is used to extract the reference to the procedure being invoked at the callsite. This procedure reference is a reference to the procedure's placeholder. The program tree location of procedure references, callsites, and debug statements must be uniquely identified because their program tree reference is used for correction of the runtime stack. For random runtime errors an approximation is given. Examples of approximations are the program line, statement, or expression containing the runtime error.

5.2.1.2. Maintenance of Mapping Information

As mentioned earlier, the common way of providing mapping information is the generation of a mapping table by the compiler. This mapping table supports mapping of program locations in both directions equally well. A reference is mapped by a search of the table. The table lookup is improved by organizing the information into subtables, one for each procedure, and a table of procedure references. The lookup now amounts to first locating the procedure containing the reference, and then searching the procedure's subtable.

An extreme case of a structured mapping table is obtained by embedding the object code references in the program tree. The derivation of object code references from program tree references is trivial. The reverse mapping, however, requires a search of the program tree. The search can be guided if both the object code location and the object code size of the given subtree are available to the search. Search of subtrees not containing the reference is avoided at the cost of adding the size information to every node in the program tree.

The opposite extreme is the implementation of the mapping table in the execution image. Program tree references are inserted into the object code at specific places. A similar practice is occasionally found in existing debugging systems. The Bliss debugger Six12, for example, stores a key in front of the procedure object code to get quick access to the mapping table. This form of mapping information maintenance violates LOIPE's premise to only burden the execution image with overhead for actually used debugging support.

The last alternative is to not maintain mapping information explicitly. The code generator has produced the original mapping. Therefore, it is able to reproduce the mapping as long as the processed source program has not been modified. Both program tree references and object code references can be mapped.

5.2.1.3. LOIPE's Hybrid Mapping Approach

LOIPE uses a combination of methods for maintenance of mapping information, trading off the cost of maintenance with the cost of lookup for different types of references. LOIPE must be able to map procedure references, callsite references, debug statement references and locations of runtime errors. Object references do not have to be mapped because they are never shown explicitly to the user. Procedure, callsite and debug statement references are mapped every time the callstack is displayed. Note that the program tree representation of the callstack can be updated incrementally. If the object code references are stored with entries of the callstack, LOIPE can quickly determine the callstack entries that have not changed since the last snapshot. The debug statement reference appears only as the current excution point of the top procedure on the callstack, as does the location of a runtime error. Runtime error locations are mapped infrequently because execution suspension due to a runtime error should be rare, especially with a relatively safe programming language. The callstack program tree is updated before any modifications by the user are permitted. Thus, all control flow references are available as program tree references for correction when procedures are replaced due to modifications.

For procedure references, a mapping table is maintained that parallels the placeholder space, i.e., the entry vector for procedures. An entry in the table contains the physical location of the procedure's object code (as does the placeholder of the procedure), the size of the procedure code, and the program tree reference to the procedure subtree. The placeholder address of the procedure can be used as a key to the mapping table, because the mapping table parallels the placeholder space. This placeholder address is available both in the execution image as the procedure reference at the callsite, and in the program tree as part of the information kept in the symbol table entry for the procedure identifier. Thus, the mapping in either direction is a matter of indexing into the mapping table.

In addition to supplying mapping information for procedure references, this table permits localization of a random object code reference. A reference is localized by checking whether it falls into the range covered by the physical base address and size of any given procedure.

The cost of localizing the right procedure is in the worst case linear to the number of procedures in the program (linear search).

The mapping information about all callsites in a procedure is kept in a mapping table associated with the procedure. This representation is preferred over storage of the object code address with the callsite node because the number of entries in the callsite mapping table is less than the total number of nodes in a procedure, resulting in a quicker reference lookup. The lookup cost for the object code reference of a callsite consists of the cost of localizing the reference to a procedure and the cost of searching the callsite mapping table of the localized procedure. This lookup cost could be greatly reduced (to two indirect references) if we were willing to pay the cost of storing one node reference with the callsite in the execution image. This latter solution was not adopted because it imposes space overhead on the execution image for the sake of debugging. Object code references of callsites are corrected in the runtime stack by mapping the program tree reference in the callstack to the new object code reference with the newly generated callsite mapping table.

For debug statements LOIPE includes a program tree reference in the code being generated for a debug statement. Because debug statements are inserted into the object code only when they are enabled, the size of the execution image is not increased by program tree references for unused debug statements. When execution is suspended at a debug statement, the program tree reference is passed back to LOIPE. It provides immediate access to the source program location of the debug statement.

The point of suspension in the debug statement object code is recorded on the runtime stack. It must be corrected when the procedure containing the active debug statement is being replaced. The new location in the object code is determined by the code generator as part of the replacement process. The replacement of the procedure may have been due to the removal of the active debug statement. This fact is passed on to the code generator. In that case the code generator locates the object code location of the succeeding statement as the location to be resumed.

LOIPE maps the object code location of random runtime errors into program tree reference through the use of the code generator. First, the procedure containing the object code reference is located through the procedure mapping table. Then, the procedure subtree is submitted to the code generator for reprocessing. The code generator is able to reproduce

the mapping because the program part has not been modified yet. The code generator returns the node in the program tree that it considers as most closely reflecting the location of the runtime error.

In summary, LOIPE is able to provide all information necessary for mapping the control flow. Explicit mapping information is kept in a procedure mapping table, and a callsite mapping table per procedure. The procedure mapping table is also used by the partial replacement mechanism. Other mappings are established by the code generator. The cost of updating the program tree representation of the callstack is acceptable especially because the tree representation of the callstack is updated incrementally.

5.2.2. Access to Data Objects

One of the tasks of a debugger is to provide access to the contents of data objects, both for display of the current value and for its modification by the user. It must convert the representation of the object in the execution image into a human readable form and vice versa. Some existing debuggers show data objects in terms of the data abstractions that are specified in the program. For that purpose, symbol table information on the data type is usually made available by the compiler and interpreted by the debugger.

In LOIPE, a placeholder subtree is maintained for the current value of a data object. This additional subtree is associated with the program tree representation of the declaration (see section 3.2.2). It is generated automatically by the system, which interprets the type definition of the data object. Regeneration is not necessary unless the declaration or the data type of the object have been modified.

One possible way of providing access to the current value of a data object is the following. Whenever a display request is issued for a data object its current value is retrieved from the execution image and inserted into the placeholder subtree. As discussed in section 3.2.2, the unparser of the structure editor then generates the textual representation, converting the internal representation of the base type elements into their external representation. In the case of a composite type, each field of the current value subtree must be supplied with values from the execution image. This is illustrated in Fig. 5-1.a. The retrieval mechanism must have access to type information in order to extract the component values correctly.

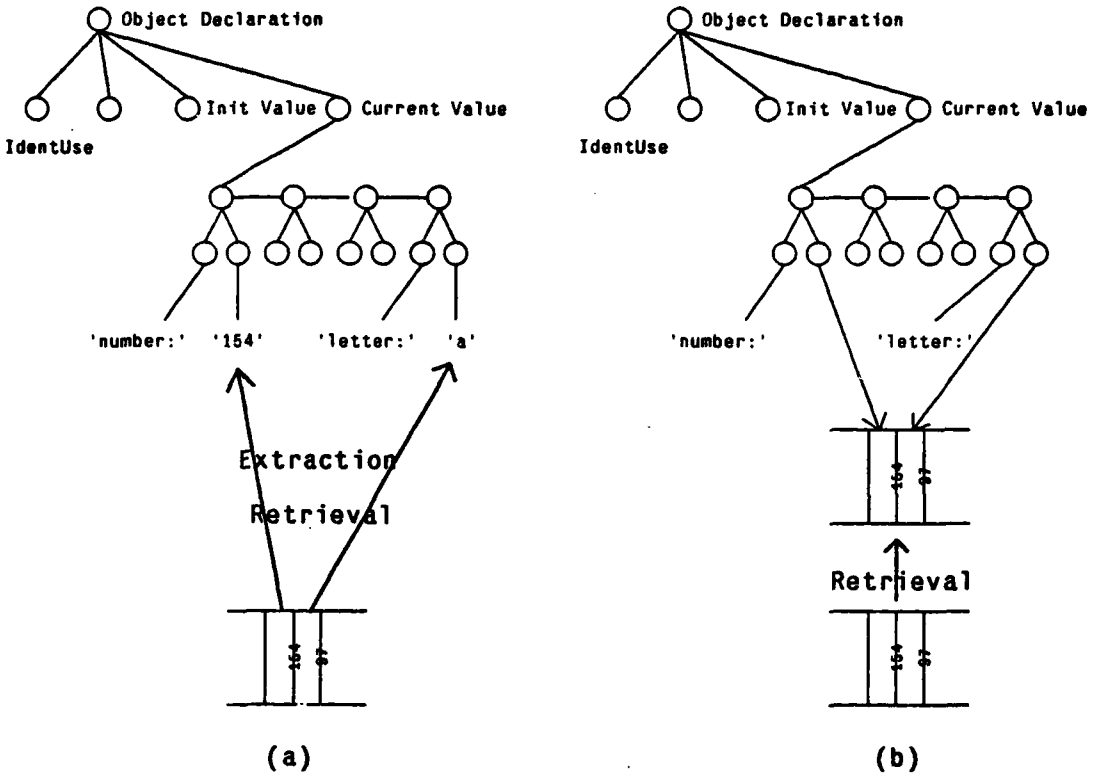


Figure 5-1: Retrieval and Display of Current Values

LOIPE has adopted a different solution that utilizes the unparser for the extraction of component data. The unparsing mechanism is extended to support indirect access in addition to conversion of the internal representation of base type values. (The support of indirect access is already available in the structure editor for display of strings.) Using the indirect access mechanism the unparser can extract the components of a data object from contiguous storage, which is referred to by the placeholder subtree (see Fig. 5-1.b). This contiguous storage is either a buffer into which the data object has been retrieved from the execution image, or a range in the LOIPE address space into which the execution image is mapped.

Local data objects and actual parameters of a procedure invocation are retrieved in one block because they are located together in the procedure activation record. The unparser extracts individual local variables out of this contiguous data area. In LOIPE,the user can examine the local variables and parameters of only one active procedure at a time. Therefore,

the same contiguous storage area can be used for retrieval of all local data. This allows the references in the placeholder subtree to be bound at the time code is generated for the enclosing procedure, i.e., when the offsets in the activation record are assigned. The cost of displaying a global data object or the locals and parameters of a procedure amounts to a block retrieval of the data from the execution image and the unparsing of the appropriate subtree with the *current value unparse scheme*. Measurements on the prototype implementation of LOIPE indicate that the response time for display requests is acceptable (see section 6.1.2).

In LOIPE the current content of a data object can be modified by editing the placeholder subtree. The structure editor converts the new values into their internal representation and stores them with the placeholder subtree, i.e., buffer area assigned to it. Semantic actions recognize the fact that the current value has been modified and cause the data in the buffer to be written back into the execution image.

5.2.3. Summary

We have shown that LOIPE is able to generate snapshots of the program state in the program tree representation. The program tree representation *is updated incrementally* at points of execution that are specified by the user. The code generator provides mapping information for mapping of the control flow. The amount of explicitly kept mapping information is minimized. For the display of current data object values the interpretive nature of the unparser is utilized. The user is able to modify the program state in the execution image by editing its program tree representation.

5.3. Code Optimizations and Debugging

A code generator processes the source program in sequence, generating object code instructions for each source construct one by one. An optimizing code generator attempts to improve the quality of the resulting object code by generating code for each source construct in the context of surrounding constructs. Analysis of the data flow and control flow semantics of the source program as a single static entity provides the *optimization context*. With this context information the optimizing code generator can take advantage of special hardware features that are not directly reflected in the programming language. The optimizing code generator can use this context information to determine a more efficient evaluation order than the one given by the source program.

Source program debuggers are confronted with the problem of showing the actual flow of control in the object code in terms of the source program. This task is difficult if not impossible for optimized object code. The optimizing code generator may have chosen an evaluation order that is in conflict with the order indicated by the source program, i.e., a sequence of individual constructs. Therefore, existing source program debuggers have limited themselves to nonoptimizing code generators.

A closer look at compilers, for which source program debuggers are provided, reveals that some of them actually do perform some optimizations. One example is the Unix C compiler and sdb debugger [Unix 81a]. The C compiler performs constant folding, even though the optimization flow is disabled for the use of sdb. As we shall see in the next section, this optimization does not destroy the source program ordering of the program locations that are mapped by the debugger. The control flow sequence through these program locations in the object code is identical to the corresponding ones in the source program.

A source program debugger modifies the optimized object code by patching in debug statements. Execution can be resumed because these patches are performed at program locations that are not affected by the optimizations. LOIPE uses the partial replacement mechanism to update the object code. A program modification may change the optimization context for the replaced procedure. This may result in a different object code sequence not only for the modified program part but the whole procedure. If that procedure is active the program execution site on the runtime stack is affected as well, possibly preventing continuation of execution even for enabling of debug statements.

Section 5.3.1 discusses the effects of various optimizations on the ability to display the progress of execution in terms of the source program and to resume execution after debug interactions or program modifications. Section 5.3.2 follows with a proposal for LOIPE to deal with optimizations by providing degrees of debugging.

5.3.1. Effects of Optimizations

Compilers use a variety of optimization techniques to improve the quality of the object code. A number of the optimization techniques are centered around the exploitation of general purpose registers as fast memory. A second set of optimization techniques capitalizes on improving the evaluation order of constructs as given in the source program. A

third set of optimizations makes use of static evaluation of expressions. Finally, there are some optimizations that take advantage of specific properties of the hardware instruction set.

In the following paragraphs we examine the different optimization techniques with respect to interferences with language-oriented debugging in order to determine a strategy for LOIPE to deal with them. The list of optimization techniques being discussed may not be complete but covers most of the ones commonly used.

5.3.1.1. Use of General Purpose Registers

The use of general purpose registers can both improve the execution time of the program and reduce the size of the object code. The registers are usually used within the scope of a procedure. On a procedure call the old contents of the registers are saved into the procedure activation record before they are used by the called procedure. In order to minimize the cost of a procedure call some optimizers save only those registers that are actually being used. A change in the number of saved registers affects the layout of the activation record and prevents resumption of execution.

In some languages the user specifies explicitly which local variables are kept in registers. Such a use of registers can be supported in LOIPE. The number of registers assigned to local variables is changed by the user editing local declarations. This modification results in unwinding of the callstack to remove the affected activation record, as discussed in section 3.4.1.2. As part of the removal the saved registers are restored.

Registers can act as a *cache*. The content of a data object is kept in a register for faster access. Local data objects may reside only in registers without a counterpart in main memory. The assignment of a register as the location of a local object can be made available by the code generator to permit correct retrieval of the data for debugger. The content of a global data object may temporarily reside in a register in addition to main memory. *Load-Store motion* optimization determines when it is necessary to update the copy in main memory. In different procedures the global data object may be cached in different registers. The task of locating the appropriate storage for retrieval requires detailed information from the optimization context. If the scope of the optimization context is limited to one procedure at a time, the code generator must update the main memory copy whenever control leaves the procedure. The called procedure assumes the current value to be found in the primary location of the data object. Thus, at all callsites cached global objects are written to main

memory. The debugger can retrieve the current value from there. Since the called procedure may have modified the global data object, the calling procedure must pick up its current value in main memory after the call returns.

Another use of registers is temporary storage of intermediate results. In expression evaluation the result of a subexpression may be kept in a register rather than on an evaluation stack. If the same expression is repeated in the source program, the result of the first evaluation can be retained, avoiding redundant calculation (*common subexpression elimination*). In either case the display of the program state is not affected, because the intermediate result is not visible at the source program level.

Resumption of execution cannot be guaranteed, because of possible reassignment of registers or a change in the number of used registers. This is the case when the optimization context has been affected through a program modification. Reassignment invalidates the program execution state if the registers contain valid information at current execution points. This is the case for expression evaluation if the current execution point of an active procedure is part of an expression. Similarly, the program execution state in registers can only be affected if the scope of a common subexpression contains a current execution point. A callsite may be included in the scope if a common subexpression consists of data objects that are not accessible by other procedures.

5.3.1.2. Evaluation Order

The source program specifies an initial ordering of evaluation. The evaluation ordering in the object code may differ from the initial order for two reasons. First, the semantics of many languages permit the mathematical laws of commutativity and associativity to be applied to expressions. The code generator may reorder expressions to achieve a better object code sequence. Second, the program can be viewed as a mapping of input variables to output variables. Code may be moved without affecting the program behavior according to this view.

Reordering of expressions does not affect the display of the program state, because it only shuffles the order of intermediate results. Expression reordering does affect the trace of control flow. However, the object code order may be visible to the user, even if only statements are traced. For example, expression reordering may interchange the order of invocation of two functions in an expression. Tracing of these two functions will make the interchange evident to the user. But, as mentioned above, the language does not allow the user to rely on the source program order within expressions.

Code motion affects the display of program state in two ways. First, code motion may change the evaluation order of statements, i.e., constructs that modify the program state. After the evaluation of one statement the preceeding statement may not have been executed. On examination of data object contents, the user cannot relate the shown program state to the source program. A trace of the control flow reveals the actual evaluation order. Second, code may be moved out of the branches of a conditional statement, merging two code sequences in the source program into one in object code. The moved code may include a callsite. This callsite cannot be shown uniquely as part of the callstack display, because one object location represents two callsite source code locations.

The evaluation order in the object code is determined through the optimization context. Any modification to a procedure affects its optimization context and may result in a different evaluation order. If the modified procedure is active, resumption of execution cannot be supported. Execution cannot be resumed at the current execution point, because constructs that succeed the current execution point in the old object code order may have been moved in front of the current execution point in the new order, and they would not get evaluated.

5.3.1.3. Static Evaluation

The result of constant expressions is known statically. The code generator, thus, does not have to produce object code for their evaluation. This optimization is known as *constant folding*. Since LOIPE does not support tracing of individual elements in an expression, neither the display of program state nor resumption of execution are affected.

Constant propagation is an extended form of constant folding. In constant propagation the flow of constant values through data objects is analyzed to locate data objects with statically known values for constant folding. The content of a data object can be changed with the debugger when execution is suspended. Thus, if the scope of the constant propagation includes a current execution point, the change of the current value may violate the assumption made for the propagation.

Static evaluation of expressions may show that some code is never executed. For example, if the condition of an *if* statement is known statically, one branch is never executed. In such a case, there is no need to generate object code for it (*dead code elimination*. This does not affect the display of the program state, because code that is not executed is not traced. Due to a modification, a sequence of code may become dead and will not appear in the object

code. That code sequence may contain a current execution point. According to the static view of the modified program this current execution point is not executable, i.e., the program state becomes inconsistent, and the execution cannot be resumed.

5.3.1.4. Summary

Optimizations increase the complexity of the mapping between the source code and the object code. This mapping must be understood by the debugging support in order to display the program state in terms of the source program. The code generator must provide an exact mapping for control flow and data objects. Locations of data objects are recorded in the program tree as they are assigned by the code generator. The possibility of a second object location in a register must be coped with. Exact mapping information for current execution points can be provided because the optimizations mentioned in the previous paragraphs are performed on the program tree representation before the object code is emitted. An evaluation order different from the source program order is revealed to the user through tracing of control flow. Some code generators also perform a set of optimizations directly on the object code sequence, known as *peephole* optimizations. Since these optimizations are usually performed without reference to the program tree, the provision of mapping information is more difficult.

Optimizations affect resumption of execution. Both change of a current value and modification to the source program can result in inconsistent program execution state. Change to the current value may be in conflict with the data flow analysis of the optimizations. Source program modifications change the optimization context. This may result in a change of the number of used registers in a reassignment of registers, or a different evaluation order of constructs. The side effects cannot be corrected if the number of used registers is saved, or if a current execution point is involved in an optimization.

5.3.2. Cooperation of Debugging and Code Optimization

LOIPE is an integrated environment in which different subsystems have knowledge of each other and are tailored to the common task. In the case of an optimizing code generator, both the code generator and the debugging support are aware of the effect of optimizations. The two subsystems cooperate by adjusting to each other's need. The result is the support of different degrees of debugging and optimization. The user is in control of the degree of

optimization. LOIPE allows the user to specify selectively where to use code optimizations and to indicate the degree of optimization. The degree of optimization influences the amount of debugging support available for the given program part.

5.3.2.1. Selective Use of Code Optimization

The selection of program parts for optimization works similar to the selection of program parts for code generation in the Interlisp system. The user selects program parts to be optimized through an *optimization pragma*. This pragma can be attached to a single procedure or a module. As procedures are processed by the partial replacement mechanism, they are submitted to either the nonoptimizing or the optimizing code generator. For nonoptimized procedures all debugging facilities are available to the user. For optimized procedures debugging supported is limited. The amount of supports varies according to the optimizations used in a specific code generator. No attempt is made to utilize information from the optimization context. Only those parts of the program state of an active optimized procedures that can be shown consistently are displayed.

In the callstack the reference to an invoked procedure can always be shown. The current execution point is not always uniquely identified (see above on code motion). Even though the control flow state, i.e., the callstack, is shown in terms of the source program, and optimized procedures cannot be traced, the object code evaluation order of an optimized procedure is not hidden from the user. If the optimized procedure calls two procedures being debugged, the reordering of their invocations is visible in the control flow trace.

The displayable data object state is restricted, if the full range of optimizations is applied. Only data objects that are not cached in registers can be shown. These are the actual parameters. Local variables often are not cached because a register can be assigned as primary storage location. Global data objects are only shown, if caching is not performed, or a callsite causes update of the primary object location under the assumption that the called procedure potentially accesses the object. Support for change of the current value of a data object is restricted in the same manner.

For modifications to optimized procedures no attempt is made to support resumption of execution at the point of suspension, if the procedure is active. Instead, the callstack is unwound to allow continuation from the calling procedure. Enabling of any debug statement in an optimized procedure does not only cause unwinding, but also results in nonoptimized

code generation. That is to say, the optimization pragma for a procedure is overwritten by any attempt to debug.

5.3.2.2. Degrees of Debugging and Code Optimization

LOIPE allows the user not only to indicate for each procedure whether optimizations should be used, but also how much optimization can be permitted, i.e., how much debugging is desired. In addition to *no optimization* and *full optimization*, it is possible for LOIPE to support two intermediate forms. In these intermediate forms, enabling of debug statements does not disable optimization. One intermediate form provides *complete display* of the program state. In this intermediate form the optimizing compiler is asked not to cache data objects across callsites or debug statements, i.e., to write the cached value back to the primary storage location if it has been modified.

The second intermediate form provides *full debugging* support even though the code is optimized. This intermediate form requires more consessions from the optimizing code generator. First, a change in the number of registers that is saved on an invocation is avoided by always saving a fixed number independent of the number used. Statistics shows that most procedures use only a small number of registers [Lunde 74]. Thus, saving three registers seems a reasonable compromise between the extra overhead for saving unnecessary registers and limitation of optimizations. Second, current execution points, i.e., callsites and debug statements, have a special status. They cannot be involved in any optimization. No evaluation can be moved across such a program location. No register may contain valid information, unless it is assigned to a local variable.

The assignment of registers to local variables is not dependent on the number of registers used for temporaries or as cache. They are assigned in addition to the three registers mentioned above. A change in their number results from a modification to a local declaration, which causes unwinding of the callstack (see section 3.4.2.3). The removal or disabling of an active debug statement, cannot be supported without damage to the runtime stack. The reference to the point of resumption now points into an optimized code sequence with all the problems of mapping a fully optimized procedure.

5.3.3. Conclusions on Code Optimization and Debugging

We have outlined how code optimization can be supported in LOIPE. We feel that such support is necessary. On one hand, programs never are complete. They continue to be improved and debugged. On the other hand, optimizations will continue to be used in code generation. Optimizations are necessary because the user may have written the program such that it contains redundant or unnecessary evaluations. Since the optimizer performs transformations on a copy of the program tree, many of them could be made visible to the user in the source program. It is questionable, however, whether a change of the source program by LOIPE is desirable.

When supporting an optimizing code generator in the context of a language-oriented (or source program) debugger we assume that the optimizer does not introduce errors. Since reality is different the implementors of a LOIPE must be given the ability to examine the generated object code.

5.4. Summary of the Language-Oriented Debugger Implementation

The implementation of a compiler-based debugging facility for LOIPE has been outlined. This implementation is centered around the program tree representation. Debug interactions are expressed in terms of the supported language. The implementation makes extensive use of support mechanisms that already exist for incremental program construction. The result is a simple system structure for LOIPE with a small number of additional mechanisms.

The partial replacement mechanism is used throughout the system for updating of the executable program representation. This includes the interactive insertion of debug statements. This approach has been chosen over code patching, because it utilizes existing compilation mechanisms for program construction without incurring high processing cost.

In comparison to an interpretive approach the compiler-based approach has some deficiencies. Without additional support from the hardware, complete traces of program state changes for the purpose of providing reverse execution must be implemented in software at high cost.

The program state is made accessible to the user in the program tree. The mapping between the actual program state in the execution image and the program tree is established in cooperation with the code generator. Using this mapping information, mechanisms in the structure editor extract the data for display and update it upon user modification.

The LOIPE approach of using partial replacement for the implementation of the debugger permits the use of optimizing code generators. The code generator does not have to provide extensive information of the optimization context. Through cooperation of the debugger and the optimizer it is possible to provide several tradeoffs between debugging functionality and efficency of the executing program.

Chapter 6

Evaluation of the LOIPE Design

In this chapter the LOIPE system is evaluated. The evaluation procedes in two parts. In the first part (section 6.1) a prototype of LOIPE is examined. Experiences from its implementation are presented and measurements on the prototype are compared with traditional tools for software development. In the second part (section 6.2) the generation of a LOIPE system for various languages is evaluated. First, LOIPE's ability to support Ada, an advanced high level language, is analyzed. Then, dependencies of the LOIPE design and implementation on a specific language are discussed. This chapter does not contain an evaluation of design alternatives per se because they have been discussed in the previous chapters.

6.1. A Prototype of LOIPE

In order to substantiate our claim that a language-oriented comiler-based programming environment with consistently fast response is feasible, we have implemented a prototype of LOIPE. We implemented this prototype on a VAX running Berkeley Unix. It has been implemented in GC [Feiler 79b], an extension of the C language that supports full type checking and modularity. The prototype was later extended to support program development in form of the Gandalf system [Habermann 79a].

The prototype implementation of LOIPE supports the programming language GC. The structure editor for the prototype is an ALOE for GC, which we generated with Medina-Mora's editor generator [Medina-Mora 82]. The structure editor acts as the user interface and manages the display screen of CRT terminals. A layout facility accepts user specific layout descriptions. The user moves through the program at different levels of abstraction. ALOE also maintains the program tree data base and triggers the execution of other subsystems through the action routine mechanism. We implemented semantic checking and propagation

of side effects (see section 4.1) through action routines. We have adapted the GC compiler, a modified version of the portable C compiler [Johnsson 78], to the program specific ALOE tree representation for GC. It performs complete semantic analysis and code generation at the grain of a procedure. The prototype supports incremental loading of procedures, *i.e.*, partial program replacement, and execution of incomplete programs. Support mechanisms for remote program development have been implemented, but have not been tested with a target machine different than the host machine.

The LOIPE debugger operates on the program tree and makes use of the partial replacement mechanism to handle debug statements. The prototype supports both unconditional and conditional tracing and breakpointing. Dynamic assertion checking and debug variables have not been implemented in the prototype. The idea of application scopes is shown in the example of a procedure scope for tracing and breakpointing of statements. The prototype includes display of the callstack and examination of data objects in terms of the source program. Global data objects can be monitored in the monitoring window. The display of data objects is currently limited to base types of GC and modification of the current value has not yet been completely implemented. Continuation of execution is currently provided in a limited form in that execution can be resumed at the point of suspension if the modified procedure is not in the call chain. Otherwise the callstack is unwound. The implementation of partial unwinding of the callstack is not yet completed. The global data object state can be preserved or can be reinitialized when execution is started.

With this prototype implementation we believe we have shown the feasibility of the LOIPE system as discussed in the dissertation, even though the prototype does not support all features of LOIPE. As part of the Gandalf system, the prototype has supported the development of programs consisting of seven modules with thirty procedures, but it has not yet been used extensively. Some of the ideas of LOIPE, however, have been tested out in SMILE, a system that was used to implement LOIPE and Gandalf [Denny 81, Feiler 80]. This experience as well as experience gained from the implementation of the LOIPE prototype are discussed in section 6.1.1. Our claim that an implementation of LOIPE runs reasonably efficiently is supported by measurements taken with the prototype. In section 6.1.2 these measurements are related to measurements on traditional tools for program development.

6.1.1. Experience With The Prototype Implementation

The discussion of our experience gained from the prototype implementation of LOIPE and its initial use follows the general outline of the previous four chapters. We have chosen to elaborate on those points that we think are relevant to the implementation of this particular approach. First, we discuss issues that are related to the user's view of the LOIPE system. They include the use of ALOE as the user interface, management of the display screen, the necessity of an interface to existing programs, the participation of LOIPE in the programming task, and the tight coupling of the incremental program construction and the debugging support. Then, we deal with issues resulting from the implementation of the prototype on a specific operating system.

6.1.1.1. ALOE as User Interface

All interaction with the user is implemented through ALOE. This proved to be a sound approach, even though some flaws were detected. Some problems are due to the fact that ALOE is still evolving, and the implementation is not yet completed. Other problems have been discovered only after ALOE has been used as a user interface for LOIPE and Gandalf.

ALOE's command interpreter has no lexical information on the supported language, even though the editor is syntax-directed. The user is required to use prefix format and to type the operator name for identifiers or numbers before entering the value, e.g., " + ' a int 21" instead of "a + 21". ALOE could accept infix notation for expressions without resorting to parsing if it had a more intelligent command interpreter [Feiler 81b]. The unparser of ALOE already is aware of the precedence ordering properties of parentheses (see section 2.1.4). The concrete syntax of the language is available in the unparse scheme and could be used by the command interpreter [Feiler 82].

Based on initial experience with the editor the program modification capabilities of ALOE have been improved by its implementor. Several operations (nest, unnest, transform, textual search) have been added to the basic set (construction, delete, clip, insert). However, the set of editing operations is not complete. Operations such as *substitute* or *renaming* of an identifier are still missing. Their implementation is not difficult, but every additional operation must be provided by the implementor of ALOE. Mentor [Donzeau-Gouge 80], for example, allows the user to define more complex operations out of basic ones using a meta language. Thus, ALOE to not allow the user to tailor the environment to his individual needs.

Similarly, the user cannot tailor the display of programs. Only the implementor of a specific ALOE, i.e., the person defining the abstract and concrete syntax for the supported language, has the freedom to specify the formatting of programs and the display of different abstraction levels, possibly in different windows (See section 2.2.1.1). For a full evaluation of ALOE we refer to [Medina-Mora 82].

6.1.1.2. Display Management

The display manager has been implemented as part of ALOE. This has posed some problems with managing the input and output of the program being developed. The display of the user program, which executes in a separate process, must be confined to the assigned screen area. Such support requires a display manager as an autonomous entity, which services all processes connected to a display device (see for example Canvas [Ball 81]). Unix and the I/O support through the C language are not able to support such an implementation without modifications. In the prototype implementation we were able to use a hardware windowing mechanism in the Concept100 terminal to enforce the confinement of I/O to the assigned screen area, but had to rely on the user program not redefining the hardware window.

The display management facility is limited by the use of a character oriented display terminal and a relatively low bandwidth between the computer and the terminal. The resolution of 24 lines by 80 characters restricts the number of windows that can be reasonably shown on the screen. LOIPE maps several windows that are not used simultaneously, e.g., the program, the module and the procedure window, into the same screen area. A hidden window is made visible through a keyboard command. As the number of windows increases, interaction with the window manager becomes more awkward. The user does not have visual feedback of the screen partitions because of the low display resolution.

Since LOIPE is display intensive, the communication bandwidth between the terminal and the computer is a critical factor for the effectiveness of the system. A bandwidth of 9600 baud is acceptable. As an absolute minimum we consider 1200 baud, if an intelligent display algorithm is used [Gosling 81b]. A higher resolution display, such as the bitraster display and a pointing device would improve the display management considerably, but the use of CRT terminals is feasible under the above conditions.

6.1.1.3. Interface to Existing Programs

LOIPE, as discussed so far, deals with programs that are completely represented by the program tree. Because programs already exist in text form, it is desirable to provide support for converting them into the LOIPE representation for further development with LOIPE. ALOE works exclusively with constructive commands which do not treat a program as text. To be able to work with existing programs, we built a parser which produces an ALOE program tree from program text. This parser is an adapted version of the parser in the GC compiler. Some problems were encountered with the placement of comments into the program tree. Usually, comments can be placed anywhere in the program. They are discarded by the parser of a compiler. A structure editor, however, requires that all possible locations for a comment are specified in the language description, i.e., they can be entered only in well-specified places. Since comments should be preserved when converting program text to program trees, the parser must derive from the textual position of the comment where to insert it in the program tree.

During program construction the user indicates where comments are placed in the program tree. The parser of a compiler usually throws away comments. In the context of LOIPE, however, comments must be preserved, so the parser must decide as to what construct to associate the comment with.

6.1.1.4. Active Participation

The concept of active participation exists in the SMILE system [Denny 81] as well as in LOIPE. SMILE was used extensively by all members of the Gandalf group for the implementation of the ALOE, the LOIPE, and the Gandalf system. The following list reflects those features of active participation that were valued most highly:

- The system's responsibility to map the program tree, i.e., the source program, into the existing file system; the user is not bothered with file names.
- Warning of potential damage beforehand when a modification with effects on other program parts is started; request for confirmation to continue modification.
- Reporting of semantic errors while still in context; the grain of a procedure seemed satisfactory.
- Propagation of side effects; reporting of these side effects and the ability to be guided to the location of errors.

- The ability to suppress error messages.
- Nonenforcement of semantic correctness.

These features give the programmer the necessary support and confidence to attempt modifications of globally used type definitions and procedure specifications without risking a chaos. The additional cost for maintaining the necessary semantic information for propagation of side effects in the program tree is well spent, considering the alternatives in existing systems (see section 6.1.2).

6.1.1.5. Integrated Language-Oriented Debugging Support

LOIPE's support for interactive program development and debugging is superior to traditional systems, even though the functionality of the debug statements that have been implemented in the prototype does not exceed that of debug actions in existing debuggers. The integration of program construction support and debugging support into one system allows the user to interact with both of them in the same manner. As extensions of the supported language, debug statements and program execution state blend into the source program. The user can switch between debugging and program modification smoothly without loss of the debug context. Modfications are processed incrementally, and incomplete or incorrect programs can be executed. *This provides a quick turnaround time, and important* factor for truly interactive systems.

In contrast, traditional systems require the user to explicitly leave the debugger and enter the editor via the command interpreter of the generating system, when a program modification is to be made. The debug context is usually lost. The user must locate the program part to be fixed with the editor's search command. The user interface of the editor differs from that of the debugger, requiring the user to be aware that he is communicating with the editor and not the debugger. After all program modifications are completed, the object code is regenerated by explicit invocation of the compiler and the debugger is reentered. This rather high cost of a pass through the modification cycle tends to force the user to locate as many bugs as possible in one debugging session before correcting any one of them in the source program. He has to make a conscious decision when to give up finding another bug and destroy the debug context.

6.1.1.6. The Program Tree as Central Information Depository

In LOIPE the program tree acts as the program data base in which all information concerning the program is maintained. By centralizing the information, redundant copies of information are avoided. All subsystems of LOIPE use the program tree as their primary program representation. They share the mechanism to access and manipulate this structured representation of the program. This mechanism relieves the subsystems from dealing with the storage of the program tree in the file system. The program tree is partitioned and stored in several files. The program tree is partitioned such that parts of the program tree can be loaded into the LOIPE process one at a time. When checkpointing the program tree, only parts are written out. Since Unix does not support mapping of files into a process address space, the files containing program tree partitions must be read in explicitly. This is realized with the help of the *filenode* mechanism provided by ALOE [Medina-Mora 81], and a separate symbol table for each program tree partition. Unfortunately, filenodes are visible in the program tree and must be dealt with by every system part of LOIPE that works with the program tree. This problem of maintaining the structured program representation on permanent storage is being addressed in [Notkin 82].

The subsystems share the information contained in the program tree. For example, semantic information is used for semantic analysis and code generation as well as propagation of side effects. Even though a considerable amount of information accumulates with the program tree, its size compares favorably with conventional form of program maintenance, as can be seen from the measurements on the prototype.

6.1.1.7. Incremental Program Construction With Existing Software

The incremental program construction support performs many of the same functions that a compiler does. To avoid a reimplementation we interfaced the compiler for GC to the program tree. We did so by removing the parser and replacing it with an interface module to the program tree. This interface module simulates the actions of the parser while traversing the program tree. Symbols are entered into the symbol table of the compiler and expression trees are built in a form acceptable for the code generator. Whenever the user enters a module, the compiler is preloaded with the module context, i.e., with all relevant specifications. When the user leaves a procedure after a modification, the procedure's program tree is passed to the compiler. The compiler performs a complete semantic check and generates code at the same time. Semantic errors are reported back in terms of program

tree references rather than line numbers. Given a relatively clean implementation of a compiler, writing this compiler interface is a straight forward task. Additional changes in the compiler deal with the generation of indirect procedure calls and the generation of calls to trace and breakpoint handlers.

The GC compiler produces assembly code. Consequently, we must run the Unix assembler to convert it into object code and the Unix linker to relocate the object code to the correct position in the execution image. Since the linker's job is limited to relocation, only the procedure being replaced must be processed.

The GC compiler does not provide information for handling the propagation of side effects. In the regular Unix programming support this information would be provided by the user in the form of a *makefile*. The LOIPE prototype has a set of action routines that automatically maintain the propagation information in the form discussed in chapter 4 as part of the program tree.

6.1.1.8. Remote Program Development

The LOIPE prototype executes the user process separately from the debugger process under Unix. However, LOIPE does not use the Unix *ptrace* facility to access the user address space. This mechanism provides too small a window for transfer of more than a couple of words. Instead, the LOIPE process and the user process communicate through Unix pipes in the manner outlined in section 4.2.3.

Unix provides control over the execution of the user process in two ways. First, the ancestor process could abort the user process. A copy of the user process image is created in a file for further examination. This user process image, however, cannot be modified and its execution resumed. The second mechanism is an interrupt signal which is issued from the keyboard. This signal raises an exception in every process belonging to the job on the terminal. Individual processes can ignore the signal or set up their own handler. LOIPE sets up the user process to suspend execution and report to the LOIPE process. The LOIPE process itself ignores the actual signal, but is informed of it by the runtime system of the user process. The user program cannot redefine the handling of this signal.

6.1.1.9. Support for the Debugger Implementation

The provision of mapping information is peculiar to the specific code generator being used. The code generator of the GC compiler does not keep track of the amount of code generated. However, the mapping information can be generated with the help of the assembler. The code generator emits labels into the assembly code for all locations to be mapped, and pairs of label references and corresponding program tree references into a separate data area. This data area is processed by the assembler, which resolves all the label references, but is not added to the object code in the execution image. This is similar to the provision of mapping information for the Sdb debugger with the exception that the amount of retained mapping information is much smaller. Some compilers have a code generator that counts the amount of code being generated. Such a code generator can derive the mapping information as it is counting.

Mapping information for procedure references and global data objects is maintained by the incremental loader and updated whenever a program part is replaced. For local data objects and parameters the offsets into the activation record are assigned by the compiler. Instead of simulating this assignment process, we simply submit the procedure to the compiler and query the offsets from the compiler symbol table, whenever necessary.

From the discussions in section 5.1 we have seen that interpretation has an advantage over compiled code if it comes to monitoring access to data objects and recording of changes of the program execution state. Such support requires explicit insertion of debugging code in the object code at predetermined locations. Random access to a data objects cannot easily be monitored. However, the hardware (or firmware) can be viewed as an interpreter itself. Under this view, the hardware could be expected to give support to debugging, in addition to single stepping of individual object code instructions. Until such hardware support is available, software must implement the support by inserting code at appropriate locations.

In Unix a process has the ability to save a copy of the process image in a file. This file can be examined, but cannot be reloaded to resume execution at the saved state. Other methods of saving program state for later recovery are the subject of research [Randell 75, Liskov 80]. Therefore, the LOIPE prototype does not support resetting of the complete program state to a previous point. LOIPE does support correction of return addresses in the control flow state, unwinding of the control flow state, and resetting to the initial state.

6.1.1.10. Code Optimizations

The GC compiler has an optimization switch. When enabled, it activates peephole optimizations. Even with disabled optimization switch the compiler performs some optimizations. The extent of optimizations in a compiler is often poorly documented or not at all. Therefore, the implementor of LOIPE must perform some tests to determine the range of optimizations in the code generator. [Wulf 79] discusses a test set for optimizations as part of a method to derive the quality of generated code from small test samples. Application of this optimization test indicates that the GC compiler used in the prototype performs only constant folding. Therefore, full debugging can be provided in LOIPE without difficulties.

Some optimizing compilers allow the progammer to specify optimization directives at a smaller grain than a compilation unit. The Bliss11 compiler, which is a highly optimizing compiler, accepts directives in the source program. With these directives the user can select ranges of program statements to be optimized or not. The user can also specify firewalls in a sequence of statements that guarantee that no optimization is carried across that program location. This shows that some optimizing compilers are prepared to make the concessions that are neceesary for LOIPE to provide the full range of debugging support (see section 5.3.2).

6.1.1.11. Tuning of LOIPE

A LOIPE implementation can be tuned in several respects. First, the display layout is defined in a description file and can be adjusted to utilize the resolution of the screen and the bandwidth of the display. Second, the language description for ALOE permits the LOIPE implementor to arrange the formatting of the program as well as to subdivide the program views into various levels of abstraction. Similarly, the partitioning of the program tree into separate files for permanent storage is controlled in the language description for ALOE.

Other forms of tuning require changes to the action routines. Such changes often consist of little more than invocation of an action in a different action routine, e.g., invocation of the code generator at the module node of the program tree instead of the procedure node. Examples of such tuning parameters are: selection of the different grains of error reporting, choice of modification unit, i.e., the grain at which semantic checking is performed, choice of replacement unit, i.e., the grain at which code is generated and replaced in the executable representation, and the selection of the mechanism for communication between the LOIPE process and the runtime support in the user process.

6.1.1.12. Extensibility of LOIPE

LOIPE provides an interactive programming environment that supports incremental program construction and debugging by a single programmer. However, LOIPE is not limited in its functionality. Additional services can be provided by extending LOIPE with appropriate subsystems. Performance monitoring has been mentioned in section 3.3.6 as a possible extension.

As part of the Gandalf project [Habermann 79a], the LOIPE prototype has been extended and integrated with additional subsystems for support of multiple versions of programs and support for management of documentation and multiple programmers. Such an extension was facilitated by three factors in the LOIPE approach:

- The program tree is the central depository of information, which is shared by all subsystems.
- Separation of abstract and concrete representation, allows the program tree to be extended with version control and management information through additions in the grammar description without increasing the complexity of the supported language (see section 3.1.2).
- All subsystems are driven by the action mechanism of the structure editor. Additional actions, i.e., calls to added subsystems, are asociated with the appropriate program tree nodes in the grammar description.

6.1.1.13. Summary

Even though the LOIPE prototype does not implement to full functionality of the LOIPE design, we believe that, despite the fact that the set of implemented debugging aids corresponds to those of existing debuggers, LOIPE provides more effective support for the development of even larger programs in an interactive manner. The effectiveness of LOIPE results from

- the structured and organized presentation of information to the user within the limits of the display medium,
- the active participation of LOIPE by maintaining a consistent program data base at any time and informing the user of semantic errors while the user is in context, and
- the flexibility provided by the ease of alternating between program construction and debugging and by the interactive behavior of the system in always maintaining an executable program.

The LOIPE prototype illustrates the possibility of interfacing existing software, i.e., a compiler for semantic checking and code generation. The implementation of the LOIPE prototype shows the benefit of uniform access to the program in a structured manner through the program tree. It also points out the necessity for adequate support for storing the program tree permanently. Support for recovery from system crashes must be provided, but has not been addressed in the prototype. The extensibility of LOIPE for additional functionality has been demonstrated with the Gandalf system.

6.1.2. Measurements on the Prototype

We have taken measurements on the prototype implementation of LOIPE. The results are compared to those of existing programming support for C on Unix, i.e., a screen editor (Emacs), the GC compiler, and the source program debugger Sdb. The object code size of the systems as well as processing times of different operations and the storage cost for programs are compared. These numbers, we believe, show the feasibility of the LOIPE approach as an interactive system. No measurements have been taken to verify an improvement of the development cost of a program in LOIPE over traditional systems. Such a study requires human factors research and is beyond the scope of this dissertation.

6.1.2.1. System Size

The system size is the amount of space it takes to run a programming evironment. Both the code size and the combined size of code and initialized data such as compiler tables and the grammar for ALOE have been measured. The measurements do not include the space for the source program in core, i.e., the program tree in the case of LOIPE and the program text in the case of Emacs. Also excluded from both measurements is the cost of the Unix assembler and linker, because the two are executed as separate processes in both programming environments.

The measurements are given in Fig. 6-1. In these measurements LOIPE compares favorably to the Unix support. Both editors use the same display optimization package, which keeps two copies of the screen core resident. The large difference between code size and total size for both compilers is mostly due to static allocation of the compiler's symbol table and expression tree space. In the case of LOIPE no attempt was made to reduce that size or to avoid two symbol tables (ALOE maintains the second symbol table). As expected, the cost of

C/Unix

Bytes	Code	Code & Data
emacs	129024	176236
gc compiler	71680	176100
sdb	53248	64912
total	253952	417248

Loipe

Bytes	Code	Code & Data
Aloe	69372	127260
Grammar	-	10840
Parser	12370	23146
Compiler	51764	145826
Propagation & Incr. Loading	14572	21926
Debugging	4356	7652
total	152434	336650

Figure 6-1: System Sizes

implementing LOIPE's debugger is small because many parts of the other subsystems are being utilized.

6.1.2.2. Timing of Operations

All timing measurements were performed on a lightly loaded VAX 11/780 (two users). The measurements were taken with the a microsecond clock provided by the Unix system and represent elapsed time. Fig. 6-2 shows processing times for a procedure modification. The comparison of the replacement cost for a single procedure may seem somewhat unfair to the traditional compiler-based environment. In such an environment the user frequently makes several modifications before a whole module is compiled. However, two facts have to be kept in mind. First, LOIPE compiles between editing operations whereas C/Unix delays all compilation until the program is run. At that point the user must expect a long delay. Second, as long as the user does not alternate between two procedures when editing, these two procedures are only compiled once. In C/Unix the module containing the procedures (including procedures that have not been modified) is compiled.

No measurements are available for editing operations. From our experience with using ALOE as a standalone we conclude that the higher processing cost of ALOE due to unparsing after a modification is not noticeable at 1200 baud, even if eight people use ALOE at the same time. The I/O bandwidth is the bottleneck. At 9600 baud the difference is hardly noticeable unless the system is heavily loaded.

	C/Unix	Loipe
Specifications	N/A	0.25 sec
Compilation	2.4 sec	0.2 sec
Replacement	8 sec	2 sec
Modification Cycle	12 sec	2 sec

Figure 6-2: Operation Times

The cost of a procedure modification was measured several times on a three procedure program, where the modified procedure consisted of twenty lines. The measured times do not include the time for editing. *Compilation time* is the time for producing assembly code for the modified procedure. It includes semantic analysis. The *replacement time* measures the time it takes to restore the execution image after completion of the modification, i.e., compilation time plus execution time of the assembler and linker. The *modification cycle time* also takes the context switching time between editor and debugger into account.

In LOIPE, the compilation time consists of specification processing and of semantic analysis and code generation. Specification processing is performed only when the module being modified is switched or a specification has been modified. The processing time is dependent on the number of specifications, thus must be extrapolated for larger programs. The time for semantic analysis and code generation is typical for a procedure and independent of the size of the program. In C/Unix, the compilation time measures the time it takes to process the modified procedure and the specifications of the other procedures. Note, that usually several procedures reside in a compilation unit and are compiled together.

LOIPE's replacement time shows the cost of replacing a procedure, i.e., semantic analysis, code generation, assembling, relocation and incremental loading. This replacement time does not change much when the total program size increases. The replacement time for C/Unix consists of compilation and assembling of one procedure and reconstruction of the execution image by the linker. This number increases considerably as the programs get larger due to the cost of reconstruction. Note, however, that usually several program modifications are made in C/Unix before the program is relinked.

In LOIPE there is no overhead for switching between editing and debugging. Therefore, the modification cycle time is the same as the replacement time. In C/Unix, we must account for invocation of Emacs and reinitialization of Sdb with symbol table and mapping information. The measured times do not include the cost of setting up the debug context again. The initialization time for Sdb increases with the size of the program.

The LOIPE debugger inserts breakpoint code by partial replacement. Thus, the cost of setting a breakpoint is equivalent to the replacement time of 2 seconds. Sdb's cost of setting a breakpoint is well below a second, even though we do not have concrete numbers. Display of individual data objects is virtually without delay on both debuggers. For retrieval of larger amounts of data the one-word communication path of ptrace, which is used by Sdb, can become a bottleneck.

The final measurement of the LOIPE debugger is the cost of statement tracing. The measured time of 0.4-0.7 seconds accounts for the execution of one statement in the user program and updating of the callstack, monitored data object and the cursor position for the control flow. It also includes four context switches of Unix processes, two for the execution of the statement and two for the retrieval of the monitored data object. The variance of this measurement is due to the changing amount of screen that must be updated.

In summary, the timings show that incremental program construction can be provided with fast response. On one hand, an increase in the specification processing time for larger programs has to be expected. On the other hand the use of a code generator that produces relocated or position independent (see section 4.2.1) code eliminates the cost of running the Unix assembler and linker. The cost of debugger interactions are roughly comparable with those of the Unix debugger. LOIPE cannot compete in continuous tracing with interpretive systems like the Cornell Program Synthesizer because of the context switching overhead.

6.1.2.3. Program Storage Cost

Program storage cost refers to the cost of keeping a program in the file system. Measurements on a seven module/thirty procedure program indicate that it takes about three (3) times the space for storing the program trees than the storage of program text. Note, however, that the program tree storage includes symbol tables, semantic information for propagation, and physical information regarding the location of procedures and data objects.

To make the comparison fairer, we have to consider the total amount of information to be kept for the development of a program. In C/Unix, we have to add to the program text cost

- the cost of a makefile, i.e., a file that describes the dependencies between modules and is used by the *make* facility [Unix 81a],
- the cost of an object code file per compilation unit, that contains object code symbol table information, mapping information for the debugger, and relocation information for the linker,
- the cost of an executable file, that contains the execution image and information for the debugger.

The object code is duplicated in the object code file and the executable file. The total size of an object code file is a factor of 4-5 of the size of the contained object code. The large total size is due to relocation information and mapping information. The total size of the executable file is 2 times the size of the execution image for small programs (3 procedures) and up to 8 times the size for large programs, e.g., a standalone ALOE. In LOIPE, we only have to add the cost of the execution image. All other information is already available in the program tree. Thus, the total storage cost for LOIPE of three (3) times program text plus execution image compares favorably to C/Unix's total storage cost of program text plus approximately eight (8) times the size of the execution image (four for object code file and four for executable file).

6.1.2.4. Summary of Measurements

The measurements indicate that LOIPE is a viable alternative for programming environments. The overall system size has proven to be smaller than a comparable set of software tools as a result of integration and information sharing. Information sharing through the program tree also reduces the storage requirements for maintenance fo the user programs. Note that in LOIPE that various subsystems are invoked more frequently than corresponding tools in C/Unix, but they have less information to process, due to the

incremental nature of the system. Stepwise processing of program parts by all subsystems allows LOIPE to show a truly interactive nature for all programming activities, as the timing results indicate. It is our guess that stepwise processing throughout LOIPE, e.g., partial replacement and use of available semantic information, e.g., warning of potential damage and guidance to errors, reduces the amount of computing resources used in the development of a program. This claim will have to be supported by appropriate experiments with LOIPE.

6.2. LOIPE: A System for Generating Environments

So far we have discussed LOIPE without reference to a specific programming language. In fact, LOIPE can support a whole class of languages. Even though we have been referring to abstract data types and the module concept, they are not essential to the LOIPE approach. For example, LOIPE can support Pascal, which does not have modules. These two concepts have been brought up in previous discussions to point out how their structuring capabilities can be carried over to LOIPE.

In this section we investigate the generation of a programming environment for a specific programming language from the framework that is provided by LOIPE. First, we determine LOIPE's limitations in the support of modern high-level languages by considering a LOIPE for Ada. Then, we examine the generation process by locating language dependent system parts of LOIPE and discussing their provision by the implementor of a LOIPE.

6.2.1. Ada: An Example of Support for High-Level Languages

Ada has been developed for the Department of Defense in an effort to reduce the number of languages being used by them. The resulting language is a powerful, high-level language that attempts to satisfy the requirements of a wide range of applications. It is, therefore, a good candidate to test LOIPE's ability to support different programming languages. In the previous chapters we discussed LOIPE with a language supporting abstract data types and the module concept in mind. Therefore, we limit the treatment of Ada to those parts that require additional elaboration.

6.2.1.1. Overloading of Operators

In most languages only one declarative instance of an identifier is legal in any scope. Ada permits the same identifier to be used for several procedures without hiding each other. Such *overloaded* operators must uniquely identifiable from contextual information at the use site. The actual identification process is discussed in the Rationale for Ada [Ichbiah 79]. The binding of a use site to the correct definition site of an operator is supported in LOIPE through the use of an appropriate name/symbol table mechanism. ALOE permits various name/symbol table implementation to be used that satisfy the expected interface specifications (see [Medina-Mora 81]).

6.2.1.2. Packages

An Ada package consists of a *visible part*, a *private part* and a *body*. The visible part provides specifications of program parts that can be used outside a package. The private part describes the representation of private data types which cannot be used outside the package, but is needed for code generation. The package body contains the actual implementation of a package. The effect of modifications of the implementation is entirely localized to the scope of the package body. Modifications of specifications in the package body only affect program parts in the same package, whereas modifications of the visible part of a package affect all packages that use the modified package. A modification to the private part of a package does not change the semantics of the visible part, but affects the physical information. Therefore, these modifications must be propagated in the same manner as modifications to the visible part. The actual propagation mechanism of LOIPE is not aware of the distinction between modifications to the visible part, private part, or body. It processes the use list of the modified specification. The semantic analyzer must know about the scope and visibility rules in Ada. When binding use sites according to these rules the correct uselists are set up.

6.2.1.3. Separate Compilation

Ada supports *separate compilation* with the goal to partition large programs into more manageable parts and to help reduce the cost of compilation [Ichbiah 79]. Separate compilation is distinguished from independent compilation in that it allows program parts to be compiled by themselves, but requires a certain compilation order to be adhered to in order to enforce interface checking between compilation units. Ada's compilation order and recompilation order are reflected in LOIPE's propagation mechanism. LOIPE propagates new

specifications as well as modifications to existing specifications as they are entered by the user (see section 4.1.3).

Ada requires the user to make decisions as to the partitioning of a program into separate compilation units, i.e., into files. For that purpose it provides a stub construct called *separate*. In the context of LOIPE there is no need for such stubs. LOIPE is responsible for maintaining the program in files and for submitting it in appropriate chunks to the compiler, relieving the user from that task. Stubs are also not necessary for improvement of program readability, because LOIPE already provides mechanisms to show the program at various levels of detail.

6.2.1.4. Exceptions

Ada provides an exception handling mechanism that terminates the operation which raised the exception. This means that the runtime stack is unwound to the scope of the appropriate handler. Exception handling does not pose problems to LOIPE. Exception handlers are treated like other parts of the program, i.e, can be debugged. It is expected from the runtime system of Ada to report any exceptions, that are not caught by the user program, to LOIPE as outlined in section 4.2.3. It may be desirable to display to the user the set of handlers that are able to process exceptions at any given point in execution in a manner similar to the display of the callstack.

6.2.1.5. Generics

Generics provide a facility for translation time parameterization of program units. A generic program unit definition acts as a template for different instances without requiring replication of the source code. Different instances may result in separate copies of object code. In LOIPE, we have to assign a separate placeholder for each instance of a generic procedure such that the procedure can be identified correctly in terms of the program tree, e.g., for display of the callstack. The placeholders of two instances may refer to the same piece of object code if its code sequences are identical. Debug statements are defined and enabled in the generic definition. This results in reprocessing of all instances by the partial replacement mechanism. This is similar to the insertion of debug statements in inline procedures, where all procedures enclosing a callsite must be reprocessed to reflect the insertion in the execution image. The display of data objects in an active instance of a generic procedure requires additional support. The type information must be retrieved through the instance declaration, which is accessible through the callstack (see above). In summary, support of generics requires extentions to the mechanisms provided by LOIPE.

6.2.1.6. Tasking

Ada supports concurrent processing through tasks and provides a mechanism for communication and synchronization. In LOIPE, we have not addressed issues related to the development of such programs. Because the user program is executing in a separate process, we can extend LOIPE to connect to different processes and provide sequential debugging aid for each of them. However, incremental construction and debugging of a multiple task program at the source program level requires further investigation to appropriately deal with this new language concept. We are aware of only one project that addresses the problem of adequately supporting tasking in program development [Mauersberg 82].

6.2.1.7. Summary

LOIPE provides support for most of Ada. Two concepts, generics and tasking, require further attention, if they were to be supported by LOIPE. LOIPE eliminates Ada's concern for partitioning of the source program for separate compilation. The remainder of the language is directly supported by LOIPE. Thus, other languages with similar or smaller sets of constructs, e.g., Pascal, Euclid, C, or Fortran, can also be supported by LOIPE. Due to the incremental nature of LOIPE's processing of program *parts*, *forward declarations* of specifications, which are introduced into some languages to reduce the number of passes in the compiler, can be eliminated. In the next section we investigate the effort of generating the language dependent parts of LOIPE for a specific language.

6.2.2. Generation of a LOIPE

The program tree reflects the structure of the program as it is expressed by the language. Some parts of LOIPE understand the details of a specific language, whereas other parts only make use of certain aspects of the language and are independent of a specific language. We refer to the language independent parts of LOIPE as the LOIPE *framework*, from which a language specific LOIPE is generated. Knowledge of a specific language is embedded in LOIPE both in descriptive form and in the code of system parts. When building a LOIPE for a specific language the implementor must add the language description and language specific code to the language independent LOIPE framework. In paragraph 6.2.2.1, we elaborate on the generation of the language description, which defines the structure of the program tree and provides syntactic information for the language-independent structure editor, ALOE (see

also section 2.1.4 and [Medina-Mora 81]). Then, the provision of system parts with language dependent code, i.e., system parts dealing with semantic analysis, code generation, and the runtime support of the language, is discussed in paragraph 6.2.2.2. As an alternative to the generation of a LOIPE from the framework, we consider the adaptation of an existing LOIPE for one language to support a different language in paragraph 6.2.2.3.

6.2.2.1. Generation of an ALOE Language Description

Certain factors have to be taken into consideration when an ALOE language description is generated as discussed in [Medina-Mora 82]. The ALOE language description has the form of a grammar for abstract syntax of the language and a set of unparse schemes for each production specifying the concrete syntax. It provides ALOE with language specific information [Medina-Mora 81]. The abstract syntax defines the structure of the program tree and constraints the set of legal offsprings of a node. The concrete syntax defines how nodes in the program tree are shown to the user as program text.

The function of ALOE language description is similar to that of a BNF description. The BNF description is a commonly used notation for specifying the syntax of a programming language [Backus 59]. It defines the structure of the parse tree as it is generated by a parser. The ALOE language description can be derived from the BNF description by observing the difference between a parse tree and an abstract syntax tree (see also section 2.1.4). Here are some important steps:

- Keywords, separators and terminators are not represented by nodes in the abstract syntax tree, i.e., they do not appear in abstract syntax productions; they are attached to productions in the form of an unparse scheme.

- BNF productions, whose purpose is to permit unique recognition of the concrete syntax, can be eliminated. For example, *simpleexpression*, *term*, and *factor*, which aid in the recognition of operator precedence, can be collapsed into *expression*. In a structure editor the application order of operations is indicated by the user through the construction order.

- Sequences are expressed as such in the abstract syntax without concern of left or right recursiveness of a production.

The resulting abstract syntax description is more compact than the BNF description. An example of an abstract syntax description is the structural information part of the DIANA description for Ada [Diana 81].

A language can have several possible abstract syntax descriptions. Two abstract syntax descriptions can differ in the depth of the abstract syntax tree they are describing. For example, a procedure declaration can be represented as

```
(1) procedure  => identifier    PARAMETERS    body
    body       => LOCALDECLS    STATEMENTS
```

or as

```
(2)  procedure => identifier PARAMETERS LOCALDECLS STATEMENTS
```

When deciding alternative abstract syntax descriptions the implementor must keep the following factors in mind:

- The structure of the abstract syntax tree (to be exact, the visible part as indicated in the unparse scheme) is noticeable to the user through cursor movement. Therefore, the abstract syntax grammar should avoid the creation of unecessary nodes, but express conceptual units that are present in the language. Example (1) above expresses the notion of procedure specification and body better, even though an extra node will be created in the tree that must be passed through with the cursor.

- The abstract syntax description defines the set of legal offsprings for each node. The description can be adjusted to be more restrictive, e.g., limit the set of legal offsprings for a condition to relational operators rather than permit the full set of expressions. Some restrictions may not be desirable because they result in nonuniform enforcement. In the example of relational expression, the *structure* editor enforces the use of relational operators. However, the user can still insert a variable of the type *string*.

- The abstract syntax description defines the structure of the program tree, which is used by all system parts of LOIPE. Some system parts may need information about the program that can easily be provided without burdening the user. An example is the distinction between the definition site and the use site of an identifier by two different productions (see also section 4.1.1). The distinction is not noticeable to the user because both productions use the same synonym (command name by which the production is recognized). This is only possible as long as both productions do not appear in the same legal set.

In the appendix we have given the abstract syntax description and the unparse schemes for the concrete syntax of the language GC. The GC description is the result of its extensive use as language description for an ALOE editor of GC, which is part of the SMILE system, for the LOIPE prototype, and for the Gandalf system.

6.2.2.2. Language Dependent System Parts

The ALOE language descripton provides ALOE with information on the abstract and concrete syntax of the supported language. It also specifies the partitioning of the program tree, whose structure is defined by the abstract syntax, into smaller units for file storage and the assignment of action routines to productions. Using this description ALOE manipulator, displays, and stores syntactically correct programs and invokes other system parts through the semantic action mechanism [Medina-Mora 82]. In addition, ALOE provides a library of language independent utility routines for the implementor of a system part [Medina-Mora 81]. The utility routines include program tree manipulation and traversal routines, routines for assignment of various program views to different display windows, error reporting support, access control support and program tree status maintenance.

Instead of ALOE's default table package [Medina-Mora 81], LOIPE has implemented its own name and symbol table package. It provides mechanisms for the maintenance of use lists and the propagation of possible side effects at the grain of procedures (see section 4.1). These mechanisms are language independent, thus, are available as part of the LOIPE framework. Semantic analysis itself and code generation are language dependent. Semantic analysis includes binding of identifiers to an appropriate definition site. Such language dependent code usually already exists in the form of a compiler. This compiler can be interfaced to the program tree as demonstrated by the LOIPE prototype(see paragraph 6.1.1.7). Interfacing includes

- separation of the parser from the compiler backend,
- an interface package to map the ALOE program tree into the compiler's symbol table and program tree,
- adjustment of the code generator to produce indirect procedure calls,
- adjustment of the compiler's error reporting mechanism to the facility provided by ALOE,
- support for generation of necessary mapping information,
- and a mechanism to produce relocated object code (if necessary with the help of an existing assembler and linker).

The interface package consists of a set of traversal routines and routines that provide access to the compiler's symbol table. The traversal routines traverse the ALOE program tree and invoke code in the compiler backend in the same way as the parser does. The symbol table

access routines allow LOIPE to determine the binding of identifiers in order to maintain the uselists in LOIPE's name and symbol table.

Compilers are usually not incremental. However, they can still be used for the generation of a LOIPE, as has been shown with the GC compiler for the LOIPE prototype. LOIPE's incremental processing is limited to the grain of statements for semantic analysis (section 4.1.2), and to the grain of procedures for code generation (section 4.2) and for propagation of side effects (section 4.1.3. This incremental processing can be realized with a nonincremental compiler in the following way. Processing of specifications is separated from compiling procedure bodies. This separtion is implemented through the traversal routines in the program tree interface package. Separate specification processing allows the compiler's symbol table to be preloaded with context information for semantic analysis and code generation. For semantic analysis at the statement grain, code generation is suppressed or discarded after generation. For languages with name scopes the compiler symbol table mechanism usually supports only removal of all symbols in a scope. Therefore, a modification to a specification requires clearing of the symbol table for the given scope and reprocessing the specifications in the scope, if the compiler symbol table mechanism is not extended to support removal of individual symbols. The driver routines for invocation of specification processing, semantic analysis and code generation are independent of the specific language, thus, are part of the LOIPE framework.

The LOIPE debugger is mostly language independent, but it has to be interfaced with the supported language. The necessary extensions to the language for debug statements and program state representation are made to the ALOE language description (see section 3.1.2). The debug statement extensions are mapped into elements of the supported language for semantic checking and code generation by routines in the program tree interface package. Routines that generate the subtrees for current value display must be adjusted to the structure of declarations and type definitions of the specific language.

Finally, the runtime support of the supported language must be interfaced with LOIPE. Space management in the user process must be coordinated between LOIPE's incremental loader and the dynamic space allocation routines of the language. Runtime exceptions must be handed to LOIPE's runtime support package in the user process which reports them back to LOIPE (see section 4.2.3). The support for retrieval of information from the runtime stack and for its correction must be adjusted to the runtime stack layout for the specific language.

6.2.2.3. Adaptation of an Existing LOIPE

An existing LOIPE supporting one language can be adapted to support a different language under certain circumstances. The feasibility of an adaptation depends on the closeness of the two languages. LOIPE can be adapted at two levels: at the program tree level, and at the level of the interfaced compiler.

Adaptation at the program tree level means that for the second language the abstract syntax of the first language is used. The abstract syntax may be more restrictive by removing productions from the list of legal productions for offsprings, but the basic structure of the program tree is maintained. The differences between the two languages are expressed in the unparse schemes defining the concrete syntax. Unparse schemes can perform simple transformations by reordering offsprings of a node for display. An attempt has been made to use ALOE for the transformation of GC programs into Pascal [Feiler 81c], whose results are discussed in [Medina-Mora 82]. In [Feiler 82] the idea of using an ALOE-like structure editor as a multi-language editor is pursued further. This approach, however, frequently leads to support of subsets of languages only because one language has constructs not present in the other. [Albrecht 80] discusses compatible subsets of Pascal and Ada and transformations between them.

Adaptation at the level of the interfaced compiler permits the abstract syntax of the second language to differ from that of the original language. However, the program tree interface package must be able to handle the differences by mapping the new constructs into elements understood by the interfaced compiler. For example, an assert statement can be mapped into a conditional statement. Additional semantic checking may be necessary. In the example of the assertion the condition is not supposed to have side effects whereas side effects are permitted in a general conditional statement. Such additional semantic analysis must be implemented either by modifying the semantic analyzer of the compiler, or by providing a separate set of semantic routines that work on the ALOE program tree directly. Differences in the abstract syntax of the two languages may include type definitions or object declarations. In that case, the routines for generating placeholder subtrees for current values of objects must be adjusted to the structure of the second abstract syntax.

6.2.3. Summary on the Generation of LOIPEs

Using Ada as an example we have shown that LOIPE can support languages of the Pascal family. Some language constructs require extensions to LOIPE. Support for multitasking rerquires further investigation.

A LOIPE for a specific language is generated by providing a syntactic description in form of an ALOE language description and by interfacing an existing compiler and language runtime system as supplier of a semantic analyzer and code generator. The ALOE language description can be derived from the BNF description of a language. The designer of the ALOE language description must be aware of the difference in functionality of the two descriptions. Existing compilers do not have to be able to process programs incrementally in order to be interfaced. A relatively clean implementation of the compiler is, however, desired to ease separation of the parser and necessary modifications to the compiler backend. As an alternative to the generation of a new LOIPE, adaptation of an existing LOIPE can be considered, if the two languages are relatively close. The adaptation of the LOIPE prototype for GC to support Pascal for use in an introductory course was investigated by the Gandalf group [Gandalf 82].

Chapter 7

Conclusions

In the previous chapters we discussed the design and implementation of a language-oriented interactive programming environment that is based on compilation technology. Chapters 2 and 3 elaborated on the user's view of incremental language-oriented program construction and language-oriented debugging. The user uniformly performs all tasks interactively through a structure editor, and LOIPE contributes to the tasks by active participation and by hiding the underlying file system and operating system. Chapters 4 and 5 discussed the realization of such an environment through the use of compilation. The implementation of all LOIPE system parts is centered around a program tree representation which acts as central information depository. The integration of these system parts permits sharing of knowledge about the supported language and programs, and sharing of mechanisms that maintain the program data base. In chapter 6 the feasibility of the LOIPE approach has been demonstrated with an evaluation of a prototype implementation, which includes measurements of a running system. This chapter concludes the dissertation by summarizing the contributions and by indicating areas that require further investigation.

7.1. Contributions

This dissertation is a feasibility study of an interactive programming environment whose implementation is solely based on compilation technology. This environment differs from existing programming environments through the following characteristics:

- Uniformity — Uniformity is present in LOIPE in three ways. First, the user has a uniform view of the source program and the program execution state in terms of the supported programming language. Other representations of the program do not have to be dealt with. Second, interaction with different tools is not distinguishable because the tools are integrated into one system and use a structure editor as their common user interface. All interaction is performed

through editor operations. The user interacts with a single interface in form of a data-driven programming model. Third, all parts of the LOIPE system share the program tree as their common program representation and information depository, resulting in a simple system structure by avoiding redundancy of information and mechanisms, and efficient maintenance of the program on permanent storage.

- Active Participation — LOIPE actively participates and contributes to the programming task in several ways. First, it hides the underlying file system and operating system from the user by taking up the responsibility of maintaining the program in permanent storage without user interaction. Second, it maintains the program tree, i.e., the program data base, in a consistent state by automatically invoking system parts as necessary and recording the result of the processing. Side effects of changes are propagated, and the user is informed of their extent. The user can concentrate on construction and modification of the program, and can count on support for keeping the program consistent without having to explicitly invoke system tools.

- Language-Oriented Programming and Debugging — The user interacts with LOIPE through language-oriented manipulation of the program data base. Both program construction and debugging are performed in that manner. The correct syntax of the manipulated structures is enforced and the semantic consistency is checked and reported, while the user is still in context. For language-oriented debugging the expressive power of the supported language is taken advantage of. The debug state is integrated into the language and high level debugging support such as dynamic assertion checking is provided. The debug functions are implemented on the program tree and mapped into the executable representation by incremental program replacement rather than implemented directly on the execution image as done with existing debuggers.

- Flexibility — LOIPE is flexibile in two respects. First, LOIPE permits programs with semantically incorrect or missing parts to be constructed, even though the supported language may be strongly typed. Such incomplete programs can be executed at any time. Second, LOIPE immediately reflects all changes to the source program in the executable representation, such that the program can be executed without delay at any time. This is achieved by consistently applying the notion of incremental update in all system parts. Because the executable representation of the program is always up to date, the transition between program modification and debugging is not noticeable.

- Compilation — LOIPE maintains the executable representation of a program through compilation and static binding, resulting in efficient execution of the program. Flexibility is maintained through partial replacement of program parts rather than reconstruction of the execution image. All modifications including debug statements are reflected in the execution image through partial replacement. LOIPE utilizes the fact that in a compiling environment the executable representation is separate from the source program by supporting remote program development, i.e., execution of programs on a machine different than the host. The consistent use of compilation and partial replacement to

reflect all modifications in the execution allows LOIPE to support optimizing code generators.

The feasibility of such an interactive programming environment has been demonstrated with a prototype implementation of LOIPE. Measurements on the prototype indicate that the LOIPE approach compares favorably with traditional systems with respect to system size, program size, and processing or response time. An evaluation of LOIPE's ability to support the programming language Ada has shown that LOIPE is able to support languages of the Pascal family. Furthermore, we have shown that LOIPE provides a framework for generating environments. Language-specific information is added to the LOIPE framework through a formal language description and through adaptation of existing language-specific code.

7.2. Future Research

In the course of this thesis we have encountered several issues that require further investigation. We briefly summarize them in this section.

LOIPE takes a new approach to program development, in which the user interacts uniformly in terms of the program structure and the system contributes to the programming task. We have done some measurements on the prototype implementation whose results are given in section 6.1.2. These measurements of system size, program size and cost of a modification, indicate that the LOIPE approach compares favorably to traditional compiler-based programming. The measurements, however, do not permit conclusions to be drawn as to the impact of the LOIPE approach on the overall program development cost. Only a comprehensive study is able to determine the factors that possibly increase the productivity of a programmer. Such factors include the use of a structure editor over a text editor, immediate feedback on semantic consistency and aids in program construction dealing only with the source program, i.e., hiding of file system and implicit application of LOIPE system parts, the uniform treatment of programming and debugging, and the effects of language-oriented debugging on the ability to detect and localize errors.

LOIPE supports continuation of execution after program modifications. For some modifications continuation is only possible if the program execution state is reset. Complete recovery of a previous execution state is only marginally supported by LOIPE. Adequate implementation of recovery requires further investigation, drawing on experience from

research in fault tolerance and on appropriate support from the hardware architecture, which "interprets" object code representation of the program.

LOIPE maintains all information about a program in a central location, the program tree. All system parts access this program data base uniformly without concern for how the structures are kept in permanent storage. The chosen filenode mechanism is not quite adequate because the partitioning of the data base into files is visible in the tree structure. Additional research will be necessary to investigate the replacement of the file system by a data base system in the traditional sense or an abstract data system as proposed in [Notkin 82].

Various attempts have been made to provide support for multitasking and communication between the tasks in programming languages, e.g., Concurrent Pascal, Modula, Path Pascal and Ada. In other systems, e.g., in Unix, such support is available through routines in the runtime system. The discussions of LOIPE in this dissertation have ignored the need for appropriate support to develop and debug such programs. As a matter of fact, we are aware of only one research activity that attempts to provide support for developing and debugging individual tasks as well as the high-level interconnection structure of a group of tasks.

Even though LOIPE supports a specific programming language, the language specific knowledge is localized. Some of the language knowledge is embedded in LOIPE through a formal description, whereas other information is embedded directly in the code of some system parts. Section 6.2 described the process of generating a specific LOIPE from a framework by providing an ALOE language description, that defines the program data base structure and the concrete syntax and by adapting existing language-specific software, i.e., a compiler. Work has been done over many years to automate the generation of compilers from formal descriptions, i.e., generation of parsers [Johnson 75], generation of semantic analyzers [Ganzinger 77] and compiler backends [Wulf 80]. The application of this work to LOIPE in the context of incremental processing and propagation of information will have to be investigated, in order to formalize and automate more of the generation of a LOIPE for a specific language or a specific machine. The generation of parsers is only relevant in the context of providing support for conversion of existing programs in text form into the structured representation of LOIPE.

Appendix A

Language Description For LOIPE

A.1. Language Description for GC

```
{gandalf}                          /* language name */

/* terminal operators */
{
INT      = {s}          | "int"  | semmark ;
SHORT    = {s}          | "short int" | semmark ;
LONG     = {s}          | "long int" | semmark ;
UNSIGNED = {s}          | "unsigned int" | semmark ;
LFLOAT   = {s}          | "long float" | semmark ;
FLOAT    = {s}          | "float" | semmark ;
DOUBLE   = {s}          | "double" | semmark ;
CHAR     = {s}          | "char" | semmark ;
USETYPEDEF = {v}        | "@S" | semmark | lexnil ;
COMMENT = {a}           | "@>@>/* @C */" | semmark | lexcomment  ;
INTCONST = {i}          | "@C" | sIniMark   ;
CHARCONST = {c}         | "'@C'" | sIniMark  ;
STRING   = {a}          | "\"@C\"" | sIniMark  ;
IDENTUSE = {v}          | "@S" | sIDUSE |  ;
IDENTDEF = {v}          | "@S" | sIDDEF |  ;
EMPTY    = {s}          | " " | semmark ;
EMPTYNAME = {s}         | " " | semmark ;
EMPTYTYPE = {s}         | " " | semmark ;
QUEST    = {s}          | "?" | semmark | | "?" ;
DEFAULT = {s}           | "default:" | semmark ;
BREAK    = {s}          | "break;" | semmark ;
CONTINUE = {s}          | "continue;" | semmark ;
VOID     = {s}          | "/*VOID*/ ;" | semmark ;
LABEL    = {v}          | "@S:" | semmark | lexnil | ":" ;
GOTO     = {v}          | "goto @S;" | semmark | lexnil;
ADRIDENT = {v}          | "& @S" | sIniMark | | "&" ;
}
```

```
/* non-terminal operators */
{
PROGRAM = <extdef> |  "@0@N@e<no items>" | semmodule ;
EXTPROC = otype procdecl lparam -f | "@xlibproc @z" ↑
        "@u0extern @1 @2(@3);@u" | semItem; /* filenode */
EXTOBJ  = otype varbl -f | "@xextobject @z" ↑
        "@u0extern @1 @2;@u" | semItem ;    /* filenode */
PROCDEF = otype procdecl lparam body -f |  "@xprocedure @z" ↑
       "@u0@1 @2(@3)@+@N@4@-@N@u" | semItem | "PROC"; /* filenode */
OBJDEF  = type varbl oinitia    -f | "@xobject @z" ↑
        "@u0@1 @2 @3;@u" | semItem | "OBJ";  /* filenode */
OBJDEFLOC = type varbl oinitia  | "@1 @2 @3;" | semmark ;
REGDEF  = type varbl oinitia    | "register @1 @2 @3;" | semmark ;
/* nameless type structures as needed for casting */
TYPENAME = type tnchain | "@1 @2" | semmark ;
TNPTR   = tnchain (14)  | "* @1" | semmark | "*" ;
TNARRAY = tnchain oicst (15)    | "@1[@2]" | semmark | "[" ;
TNCALL  = tnchain (15)  | "@1()" | semmark | "(" ;
LPARAM  = <param>       | "@0;@E " | semmark | "," ;
DECLS   = <declaration> | "@0@N@Q@N@N@E@B @N" | semmark ;
STATS   = <stat>        | "@0@N@E " | semmark ;
COMPOUND = <stat>       | "@<{@N@0@N@Q@N@<}@E;" | semmark | "{" ;
DECLCOMPOUND = ldecl lstat | "@<{@N@1@2@N@<}" | semlocal | "{D" ;
ARRDECLP        = variable oicst (15)   | "@1[@2]" | semmark | "[" ;
PTRDECLP = variable     (14)    | "*@1" | semmark | "*" ;
PARAM   = type lvarbl   | "@1 @2" | semmark | ";" ;
LVARBL = <variable>     | "@0," | semmark | "," ;
STRUCTDEFLOC = defident lcomponent | "struct @1 {@+@N@2@N}@-;" | semmark ;
ARRDECL = varbl oicst (15)      | "@1[@2]" | semmark | "[" ;
PTRDECL = varbl (14)    | "*@1" | semmark | "*" ;
PROCDECL = varbl (15)   | "@1()" | semmark | "(" ;
STRUCTDEF = defident lcomponent -f | "@xstructure @z" ↑
         "@u0struct %1 {@+@N@2@N}@-;@u" | semitemordel ; /* filenode */
UNIONDEF = defident lcomponent -f | "@xunion @z" ↑
        "@u0union @1 {@+@N@2@N}@-;@u" | semitemordel ; /* filenode */
TYPEDEF = type varbl -f | "@xtypedef @z" ↑
        "@u0typedef @1 @2;@u" | semitemordel ;  /* filenode */
ENUMDEF = defident lenums -f | "@xenum @z" ↑
        "@u0enum @1 { @2 };@u" | semitemordel ; /* filenode */
/* C oddity. Defintion of type and use in declaration in one */
NEWSTRUCT = oident lcomponent   | "struct @1 {@+@N@2@N}@- " | semmark ;
NEWUNION = oident lcomponent    | "union @1 {@+@N@2@N}@- " | semmark ;
NEWENUM = oident lenums         | "enum @1 { @2 } " | semmark ;
/* use of types */
STRUCT = ident  | "struct @1" | semmark ;
UNION = ident   | "union @1" | semmark ;
ENUM = ident    | "enum @1"    | semmark ;
LENUMS = <enumelem>     | "@0, " | semmark | "," ;
ASSELEM = defident cexp | "@1 = @2" | semmark | "=" ;
LCOMPONENT = <fielddecl>        | "@0@N" | semmark | "lC" ;
FIELDDECL = type lvarbl | "@1 @2;" | semmark | ";" ;
NONAMEFIELD = type tnchain | "@1 @2;" | semmark ;
```

```
/* statements */
CSTAT   = stat comment  | "@1@N@2" |  semmark  | "/*" ;
EXPSTAT = exp           | "@1;" | semmark | ";" ;
IF      = exp stat      | "if (@1)@+@N@2@-" | semmark ;
IFE     = exp stat stat | "if (@1)@+@N@2@-@Nelse@+@N@3@-" | semmark ;
WHILE   = exp stat      | "while (@1)@+@N@2@-" | semmark ;
DO      = stat exp      | "do@+@N@1@-@Nwhile ( @2 );" | semmark ;
SWITCH  = exp compound  | "switch ( @1 )@+@+@N@2@-@-@N" | semmark ;
CASE    = cexp          | "@<case @1:" | semmark ;
FOR     = oexp oexp oexp stat | "for (@1; @2; @3)@+@N@4@-" | semmark ;
RETURN  = oexp  (16)    | "return @1;" | semmark ;
/* initialized declarations */
PROCDECLP = variable (15)       | "@1()" | semmark | "(" ;
LCEXP   = <ccexp>       | "{@0,@Q}" | sIniMark | "{" ;
LEXP    = <exp>         | "@0,@E " | semmark | "," ;
CCEXP   = cexp comment  | "@1@2" | semmark | "/*" ;
```

```
/* expressions */
EXPIF    = exp exp exp (3)        | "@1 ? @2 : @3" | semmark | "?" ;
FIELD = lvalue ident (15)| "@1.@2" | semmark | "." ;
PTRFIELD = exp ident (15)| "@1->@2" | semmark | "->" ;
CONTENTS         = exp    (14)     | "*@1" | semmark | "U*" ;
ADDRESS = exp (14)        | "&@1" | sIniMark | "U&" ;
NEGATE   = exp (14)       | "-@1" | semmark | "U-" ;
NOT      = exp (14)       | "!@1" | semmark | "!" ;
COMPLEMENT = exp (14)     | "}@1" | semmark | "}" ;
BINC     = lvalue (14)    | "++@1" | semmark | "++X" ;
AINC     = lvalue (14)    | "@1++" | semmark | "X++" ;
BDEC     = lvalue (14)    | "--@1" | semmark | "--X" ;
ADEC     = lvalue (14)    | "@1--" | semmark | "X--" ;
SIZEOF   = typexp (14)    | "sizeof(@1)" | semmark ;
CAST     = type tnchain exp (14) | "(@1 @2) @3" | semmark ;
PROCCALL         = procexp lexp (15)| "@1(@2)" | semmark |"(" ;
INDEX    = arrexp exp (15)| "@1[@2]" | semmark | "[" ;
PAREXP   = exp (15)       | "(@1)" | semmark | "()" ;
PLUS     = exp exp (12)   | "@1 + @2" | semmark | "+" ;
MINUS    = exp exp (12)   | "@1 - @2" | semmark | "-" ;
MULT     = exp exp (13)   | "@1 * @2" | semmark | "*" ;
DIV      = exp exp (13)   | "@1 / @2" | semmark | "/" ;
MOD      = exp exp (13)   | "@1 %% @2" | semmark | "%" ;
LSHIFT   = exp exp (11)   | "@1 << @2" | semmark | "<<" ;
RSHIFT   = exp exp (11)   | "@1 >> @2" | semmark | ">>" ;
LSS      = exp exp (10)   | "@1 < @2" | semmark | "<" ;
GTR      = exp exp (10)   | "@1 > @2" | semmark | ">" ;
EQL      = exp exp (9)    | "@1 == @2" | semmark | "==" ;
GEQ      = exp exp (10)   | "@1 >= @2" | semmark | ">=" ;
LEQ      = exp exp (10)   | "@1 <= @2" | semmark | "<=" ;
NEQL     = exp exp (9)    | "@1 != @2" | semmark | "!=" ;
BAND     = exp exp (8)    | "@1 & @2" | semmark | "&" ;
BOR      = exp exp (7)    | "@1 | @2" | semmark | "|" ;
BXOR     = exp exp (6)    | "@1 ^ @2" | semmark | "^" ;
AND      = exp exp (5)    | "@1 && @2" | semmark | "&&" ;
OR       = exp exp (4)    | "@1 || @2" | semmark | "||" ;
ASSIG    = lvalue exp (2)| "@1 = @2" | semmark | "=" ;
APLUS    = lvalue exp (2)| "@1 += @2" | semmark | "+=" ;
AMINUS   = lvalue exp (2)| "@1 -= @2" | semmark | "-=" ;
AMUL     = lvalue exp (2) | "@1 *= @2" | semmark | "*=" ;
ADIV     = lvalue exp (2) | "@1 /= @2" | semmark | "/=" ;
AMOD     = lvalue exp (2) | "@1 %%= @2" | semmark | "%=" ;
ARSHIFT = lvalue exp (2) | "@1 <<= @2" | semmark | "<<=" ;
ALSHIFT = lvalue exp (2) | "@1 >>= @2" | semmark | ">>=" ;
AAND     = lvalue exp (2) | "@1 &= @2" | semmark | "&=" ;
AOR      = lvalue exp (2) | "@1 |= @2" | semmark | "|=" ;
AXOR     = lvalue exp (2) | "@1 ^= @2" | semmark | "^=" ;
COLATERAL = exp exp (1) | "@1 , @2" | semmark | "," ;
}
```

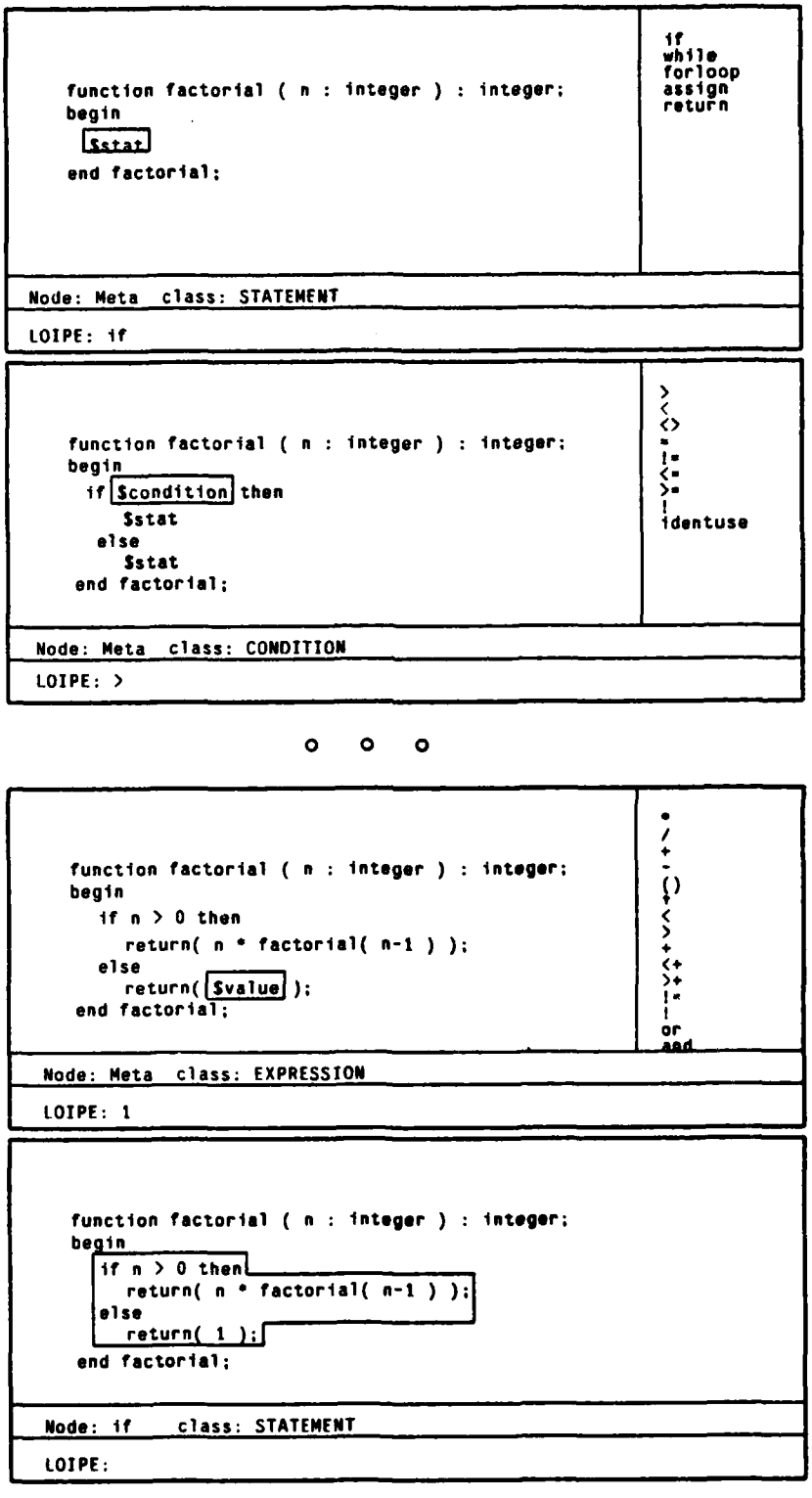
```
/* classes */

{
extdef          = EXTPROC EXTOBJ PROCDEF OBJDEF STRUCTDEF COMMENT ENUMDEF UNIONDEF
                  TYPEDEF ;
oident          = IDENTDEF EMPTY ;
ident           = IDENTUSE ;
defident        = IDENTDEF ;
lparam          = LPARAM ;
body            = DECLCOMPOUND ;
ldecl           = DECLS ;
lstat           = STATS ;
param           = PARAM QUEST ;
lenums          = LENUMS ;
enumelem        = IDENTDEF ASSELEM ;
lcomponent      = LCOMPONENT ;
fielddecl       = FIELDDECL NONAMEFIELD COMMENT ;
varbl           = IDENTDEF ARRDECL PTRDECL PROCDECL ;
variable        = ARRDECLP PTRDECLP PROCDECLP ;
procdecl        = IDENTDEF PTRDECL ;
stat            = EXPSTAT IF IFE WHILE FOR RETURN BREAK CONTINUE
                  DO LABEL GOTO SWITCH CSTAT COMPOUND CASE
                   DEFAULT VOID ;
comment         = COMMENT ;
compound        = COMPOUND ;
type            = INT CHAR NEWSTRUCT  STRUCT SHORT LONG UNSIGNED
                  LFLOAT FLOAT DOUBLE  USETYPEDEF
                  NEWUNION UNION NEWENUM ENUM ;
otype           = EMPTYTYPE INT CHAR NEWSTRUCT  STRUCT SHORT LONG UNSIGNED
                  LFLOAT FLOAT DOUBLE USETYPEDEF
                  NEWUNION UNION NEWENUM ENUM ;
typexp          = TYPENAME  IDENTUSE INTCONST CHARCONST PROCCALL INDEX PAREXP PLUS
                  MINUS MULT DIV LSS GTR EQL GEQ LEQ NEQL AND OR ASSIG APLUS
                  AMINUS FIELD PTRFIELD CONTENTS STRING EXPIF ADDRESS NEGATE NOT
                  COMPLEMENT BINC AINC BDEC ADEC SIZEOF CAST AMUL ADIV AMOD
                  ARSHIFT ALSHIFT AAND AOR AXOR MOD LSHIFT RSHIFT BAND BOR BXOR
                  COLATERAL ;
tnchain         = TNPTR TNARRAY TNCALL EMPTYNAME ;
declaration     = REGDEF OBJDEFLOC STRUCTDEFLOC COMMENT  ;
lvarbl          = LVARBL ;
oicst           = INTCONST IDENTUSE EMPTY ;
exp             = IDENTUSE INTCONST CHARCONST PROCCALL INDEX PAREXP PLUS MINUS
                  MULT DIV LSS GTR EQL GEQ LEQ NEQL AND OR ASSIG APLUS AMINUS
                  FIELD PTRFIELD CONTENTS STRING EXPIF ADDRESS NEGATE NOT
                  COMPLEMENT BINC AINC BDEC ADEC SIZEOF CAST AMUL ADIV AMOD
                  ARSHIFT ALSHIFT AAND AOR AXOR MOD LSHIFT RSHIFT BAND BOR BXOR
                  COLATERAL  ;
procexp         = IDENTUSE CONTENTS ;
arrexp          = IDENTUSE CONTENTS INDEX PLUS MINUS APLUS AMINUS AINC ADEC BINC
                  BDEC CAST ;
oinitia         = EMPTY INTCONST CHARCONST STRING LCEXP IDENTUSE ADRIDENT ADDRESS;
lexp            = LEXP ;
lvalue          = IDENTUSE INDEX FIELD PTRFIELD CONTENTS ;
cexp            = INTCONST CHARCONST STRING LCEXP IDENTUSE ADRIDENT ADDRESS ;
ccexp           = INTCONST CHARCONST STRING LCEXP IDENTUSE ADRIDENT ADDRESS CCEXP;
oexp            = EMPTY IDENTUSE INTCONST CHARCONST PROCCALL INDEX PAREXP PLUS
                  MINUS MULT DIV LSS GTR EQL GEQ LEQ NEQL AND OR ASSIG APLUS
                  AMINUS FIELD PTRFIELD CONTENTS STRING EXPIF ADDRESS NEGATE NOT
                  COMPLEMENT BINC AINC BDEC ADEC SIZEOF CAST AMUL ADIV AMOD
                  ARSHIFT ALSHIFT AAND AOR AXOR MOD LSHIFT RSHIFT BAND BOR BXOR
                  COLATERAL  ;
}
```

A.2. Abstract Syntax of Debug Statements

```
/* production for debug statements.
   Second unparse scheme hides debug statements:
   used for display of program only */

DEBUGSTATEMENT = state condition action | "@1 assert @2 do @3;" ↑ ""
                | semDEBSTMT ;
ENABLE         = {s}    | "enabled" | semenable;
DISABLE        = {s}    | "disabled" | semdisable;
PAUSE          = {s}    | "pause" | semmark;
TRACE          = {s}    | "trace" | semmark;

/* classes */

state = ENABLE DISABLE ;
condition = DECLCOMPOUND ;
action = PAUSE TRACE;
```

Appendix B

A LOIPE Session



Entering A Program

```
function factorial ( n : integer ) : integer;/* trace statement */
   begin
   enabled assert n > 0 do pause;
      if d > 1 then
         return( n * factorial( n-1 ) );
      else
         return( 1 );
   end factorial;
```

```
Node: if    class: STATEMENT
```

```
LOIPE:
```

Cursor out

```
function factorial ( n : integer ) : integer;/* trace statement */
   begin
   enabled assert n > 0 do pause;
      if d > 1 then
         return( n * factorial( n-1 ) );
      else
         return( 1 );
   end factorial;
```

```
** Error **: Variable undefined
```

```
LOIPE:
```

Replace d by n

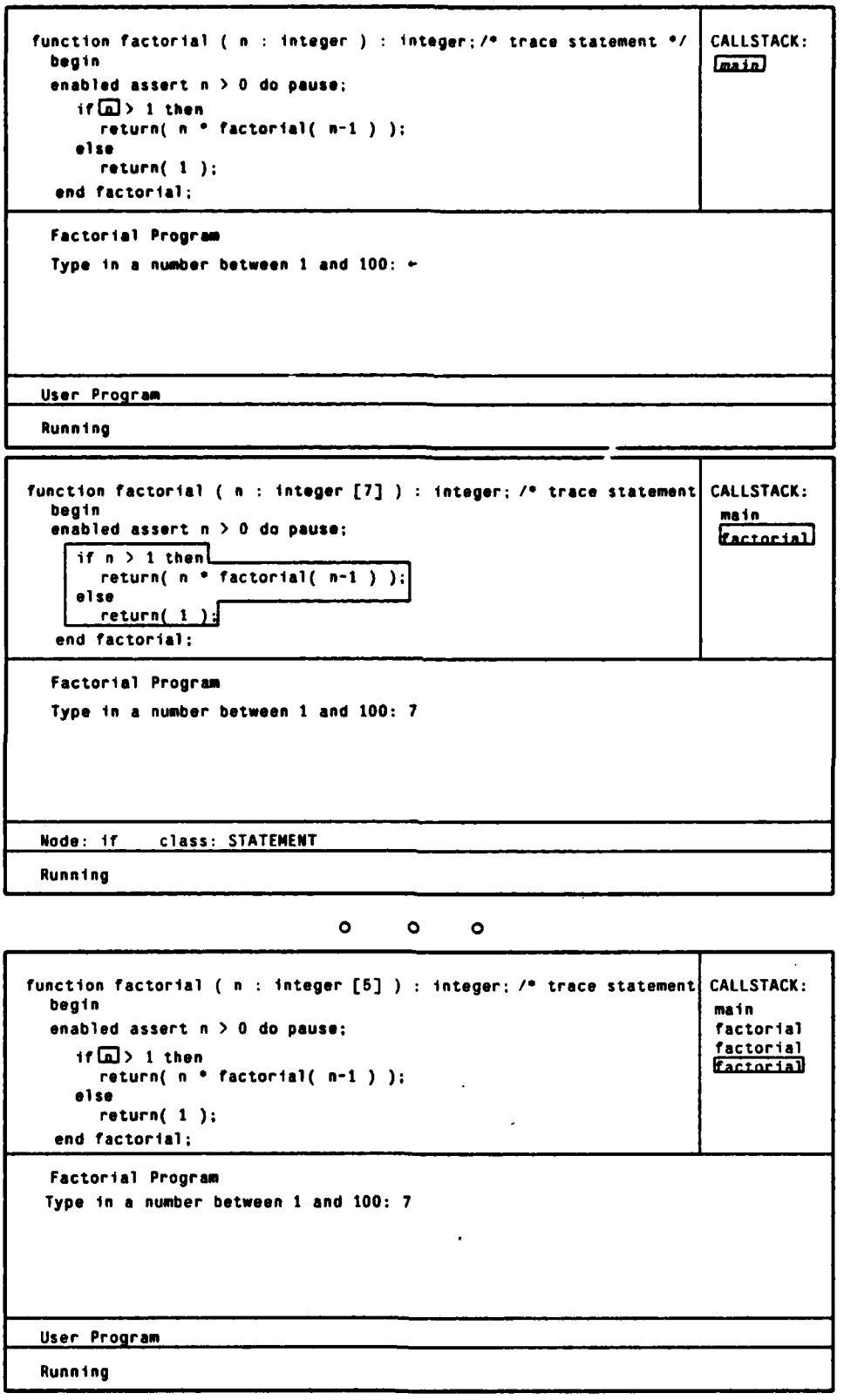
```
function factorial ( n : integer ) : integer;/* trace statement */
   begin
   enabled assert n > 0 do pause;
      if n > 1 then
         return( n * factorial( n-1 ) );
      else
         return( 1 );
   end factorial;
```

```
Node: ident class: EXPRESSION
```

```
LOIPE:.run
```

Code Generation, Replacement, and Start Execution

Detection and Correction of an Error



Trace Execution of Program

References

[Albrecht 80] Albrecht, P.F., Garrison, P.E., Graham, S.L., Hyerle, R.H., Ip, P., Krieg-Brueckner, B.
Source-to-Source Translation: Ada To Pascal and Pascal to Ada.
Sigplan Notices 15(11), Nov, 1980.

[Andler 79] Andler,S.
Predicate Path Expressions: A High-Level Synchronization Mechanism.
PhD thesis, Carnegie-Mellon University, Computer Science, Aug, 1979.

[Archer 81] Archer, J., Conway, R.
COPE: A Cooperative Programming Environment.
Technical Report TR 81-459, Cornell University, Computer Science, June, 1981.

[Backus 59] Backus, J. W.
The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.
In *Proceedings of the International Conference on Information Processing*, pages 125-132. UNESCO, 1959.

[Ball 80] Ball, J. E.
Alto as Terminal.
1980.
Carnegie-Mellon University.

[Ball 81] Ball, J. E.
Canvas: Graphics for the Spice personal timesharing system.
Proceedings of Comgraph 1981, Online Conferences , Oct, 1981.

[Barel 81] Barel, M.
Perq Pascal Extensions
Three Rivers Computer Corp., 1981.

[Belady 78] Belady, L.A.
Large Software Systems.
Technical Report RC-6966, IBM Thomas J. Watson Research Center, Jan, 1978.

[Blair 71] Blair, J.C.
An Extendible Interactive Debugging System.
PhD thesis, Purdue University, June, 1971.

[Buxton 80] Buxton, J.N.
Requirements for Ada Support Environments.
Department of Defense.

[Campbell 78] Campbell, R.H., Miller T.J.
A Path Pascal Language.
Technical Report, University of Ill. at Champaign-Urbana, Computer Science, April, 1978.

[Cattell 79] Cattell, R.G.G.
An Entity-Based Database Interface.
Technical Report CSL-79-9, Xerox Parc, Aug, 1979.

[Day 79] Day, N.G.P.
Correct Editing with ATNs.
IUCC Bulletin 1(2), 1979.

[Denny 81] Denny, B., and Feiler P.
SMILE: System Management and Incremental Language-Oriented Environment.
Manual of Gandalf project, Carnegie-Mellon University.

[Deutsch 71] Deutsch,P., and Mac Ewen.
A Flexible Measurement Tool For Software Systems.
In *Proceedings of IFIP*. 1971.

[Deutsch 73] Deutsch, P.
An Interactive Program Verifier.
Ph.D. Thesis CSL-73-1, Xerox Parc: Palo Alto, May, 1973.

[Deutsch 80] Deutsch, L.P. and Taft, E.A. (editors).
Requirements for an Experimental Programming Environment.
Technical Report CSL 80-10, Xerox Palo Alto Research Center, june, 1980.

[Diana 81] G. Goos and W.A. Wulf (editors).
Diana Reference Manual
Universitaet Karlsruhe and Carnegie-Mellon University, 1981.

[DoD 78] Department of Defense.
Requirements For The Programming Environment For The Common High Order Language.
Technical Report, Department of Defense, July, 1978.

[DoD 80] United States Department of Defense.
Reference Manual for the Ada Programming Language.
1980.
Proposed Standard Document.

[Donzeau-Gouge 80] Donzeau-Gouge, Veronique, Huet, Gerard, Kahn, Gilles and Lang, Bernard.
Programming Environments Based on Structured Editors: The Mentor Experience.
Presented at the Workshop on Programming Environments in Ridgefield, CT on June 1980.

[Fabry 76] Fabry, R.S.
How to Design A System in which Modules Can Be Changed On The Fly.
In *2nd International Conference on Software Engineering*. IEEE, 1976.

[Feiler 79a] Feiler, Peter H.
IPC System Version 1.
Gandalf internal documentation.
1979.

[Feiler 79b] Feiler, P. H. and Medina-Mora, R.
The GC Language.
1979.
Gandalf Internal Documentation. Carnegie-Mellon University.

[Feiler 80] Feiler, Peter and Medina-Mora, Raul.
An Incremental Programming Environment.
Technical Report CMU-CS-80-126, CMU, Computer Science Department, April, 1980.

[Feiler 81a] Feiler P., and Medina-Mora R.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5), Sept, 1981.

[Feiler 81b] Feiler, P.H.
Comments on the ALOE Structure Editor.
1981.
Working Notes of the Gandalf project.

[Feiler 81c] Feiler, P.H., Engholm, L.
GC to Pascal Program Translation Through ALOE.
1981.
Working Notes on an experiment.

[Feiler 82] Feiler, P.H., and Kaiser, G.E.
A Display-Oriented Structure Manipulator: A Multi-Purpose System.
Submitted to 6th International Software Eng. Conference IEEE , Sept., 1982.

[Gaines 71] Stockton Gaines.
The Debugging of Computer Programs.
Institut for Defense Analysis, 1971.

[Gandalf 82] Habermann, A.N., et.al.
A Pascal Programming Environment Supporting Pascal For Undergraduate Education.
1981-82.
Design discussions and working notes for its generation from the Gandalf system.

[Ganzinger 77] Ganzinger,H., Ripken K., Wilhelm, R.
Information Processing 77. : Automatic Generation of Optimizing Multipass Compilers.
IFIP, North Holland, 1977, .

[Ghezzi 79] Ghezzi, C., Mandrioli, D.
Incremental Parsing.
ACM Transactions on Programming Languages and Systems 1(1), July, 1979.

[Goldstein 81] Goldstein, I.P. and Bobrow, D.G.
Browsing in a Programming Environment.
In *Proc. 14th Hawaii Conference on System Science*. jan., 1981.

[Gosling 81a] Gosling, J.
Unix Emacs
Carnegie-Mellon University, 1981.

[Gosling 81b] Gosling, J.
A Redisplay Algorithm.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*. ACM SIGPLAN/SIGOA, June, 1981.

[Goulon 78] Goullon, H., Isle, R., Loehr, K.-P.
Dynamic Restructuring in an Experimental Operating System.
IEEE Transactions on Software Engineering 4(4), July, 1978.

[Habermann 78] Habermann, A.N., et.al.
Modularization and Hierarchy in a Family of Operating Systems.
Technical Report CMU-CS-78-101, Carnegie-Mellon University, Computer Science, Feb, 1978.

[Habermann 79a] Habermann, A. N.
The Gandalf Research Project.
In *Computer Science Research Review 1978-79*, pages 28-35. Carnegie-Mellon University, 1979.

[Habermann 79b] Habermann, A.N.
Implementation of Regular Path Expressions.
Technical Report, Carnegie-Mellon University, Computer Science, Feb, 1979.

[Hansen 74] Hansen, G. J.
Adaptive Systems For The Dynamic Runtime Optimization of Programs.
PhD thesis, Carnegie-Mellon University, Computer Science, Dec, 1974.

[Honeywell 80] Honeywell Inc.
Formal Definition of the Ada Programming Language.
Technical Report, Honeywell Inc, CII Honeywell Bull, INRIA, Nov, 1980.

[Ichbiah 79] Ichbiah, J.D., et.al.
Rationale for the Design of the Ada Programming Language
Sigplan Notices edition, 1979.

[Ingalls 78] Ingalls, D.H.H.
The Smalltalk-76 Programming System Design and Implementation.
In *Fifth Annual ACM Symposium on Principles of Programming Languages.*
ACM, Jan, 1978.

[Ivie 77] Ivie, Evan L.
The Programmer's Workbench - A Machine for Software Development.
CACM 20(10), Oct, 1977.

[Jensen 74] Jensen, K. and Wirth, N.
Pascal User Manual and Report.
Springer-Verlag, 1974.

[Johnson 75] Johnson, S.C.
YACC - Yet Another Compiler-Compiler.
Computing Science Tech. Report 32, Bell Laboratories, July, 1975.

[Johnson 77] Johnson, M.S.
The Design of a High-Level, Language-Independent Symbolic Debugging System.
In *Proceedings ACM National Conference.* 1977.

[Johnsson 78] Johnsson, R.K.
A Portable Compiler:Theory and Practice.
In *Fifth ACM Symposium on Principles of Programming Languages.*
SIGPLAN-SIGACT, Jan, 1978.

[Jones 78] Jones, A.K., et.al.
Programming Issues Raised by a Multiprocessor.
Proceedings of the IEEE 66(2), Feb, 1978.

[Kahn 81] Kahn, Gilles.
Beyond Mentor.
1981.
Private Conversation at Workshop on Ada Environments, Murnau Germany.

[Kay 69] Kay, A. C.
The Reactive Engine.
PhD thesis, University of Utah Dept. of Electrical Engineering and Computer Science, Aug, 1969.

[Kotok 61] Kotok, A.
DEC Debugging Tape.
Technical Report MIT-1, MIT, 1961.

[Lampson 77] Lampsom, Butler, et.al.
Report on the Programming Language Euclid.
SigPlan Notices 12(2), Feb, 1977.

[Lane 73] Lane,T., et.al.
Six12 User's Manual.
Technical Report, Carnegie-Mellon, 1973.

[Lasker 74] Lasker, D.M.
An Investigation of a New Method of Constructing Software.
Technical Report CSRG-38, University of Toronto, Computer Systems Research Group, Sept, 1974.

[Liskov 80] Liskov, Barbara.
Primitives for Distributed Computing.
Distinguished Lecture Series at CMU.
1980.

[Lunde 74] Lunde, A.
Evaluation of Instruction Set Processor Architecture by Program Tracing.
PhD thesis, Carnegie-Mellon University, Computer Science, Jan, 1974.

[Martin 77] Martin,F.H.
HAL/S - The Avionics Programming System for Shuttle.
In *Proceedings of the AIAA Conference on Computers in Aerospace.* Nov, 1977.

[Mauersberg 82] Mauersberg, H., Bruegge, B.
High-Level Network Debugging Support.
Private Communication.

[Medina-Mora 81] Medina-Mora, Raul and Notkin, David S.
ALOE Users' and Implementors' Guide.
Technical Report CMU-CS-81-, CMU, Computer Science Department, November, 1981.

[Medina-Mora 82] Medina-Mora, R.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, expected at beginning of 1982.

[Mikelsons 81] Mikelsons, M.
Prettyprinting in an Interactive Programming Environment.
Technical Report RC 8756, IBM T.J. Watson Research Center, Computer Science, March, 1981.

[Mitchell 70] Mitchell, J.G.
The Design and Construction of Flexible and Efficient Interactive Programming Systems.
PhD thesis, Carnegie-Mellon University, 1970.

[Mitchell 79] Mitchell, J.G., et.al.
Mesa Language Manual, Version 5.0.
Technical Report CSL-79-3, Xerox Parc:Palo Alto, 1979.

[Model 79] Model, M.L.
Monitoring System Behavior In a Complex Computational Environment.
PhD thesis, Stanford University, 1979.

[Myers 80] Myers, Brad A.
Displaying Data Structures for Interactive Debugging.
Technical Report CSL-80-7, Xerox Parc:Palo Alto, June, 1980.

[Notkin 82] Notkin D.
Interactive User Environments Without Files.
1982.
Thesis Proposal.

[Organik 72] Organik, E.I.
The Multics System; an examination of its structure.
M.I.T. Press, Cambridge, 1972.

[Parnas 72] Parnas, D.L.
On the Criteria to be Used in Decomposing Systems into Modules.
Comm. ACM 15(12):1053-1058, Dec, 1972.

[PDP11 73] Digital Equipment Corp.
PDP11 processor handbook.
DEC, 1973.

[Perdue 74] Perdue,C.
User's Introduction to UCILisp
Carngegie-Mellon University, Computer Science, 1974.

[Petit 69] Petit, P.
Raid.
Technical Report Operating Note 58, Stanford Artificial Intelligence Laboratory, 1969.

[Pinc 73] Pinc, J.H., Schweppe, E.J.
A Fortran Language Anticipation and Prompting System.
In *Proceedings ACM National Computer Conference*. Sept, 1973.

[Randell 75] Randell, Brian.
System Structure for Fault Tolerance.
SIGPLAN Notices 10(6), June, 1975.

[Sandewall 78] SandeWall, E.
Programming in the Interactive Environment: The InterLisp Experience.
ACM Computing Surveys 10(1), March, 1978.

[Satterthwaite 75] Satterthwaite.
Source Language Debugging Tools.
PhD thesis, Stanford University, May, 1975.

[Shapiro 80] Shapiro,E., et.al.
Pases: A Programming Environment for Pascal.
Technical Report, Yale University, April, 1980.

[Swinehart 74] Swinehart, D. C.
Copilot: A Multiple Process Approach to Interactive Programming Systems.
PhD thesis, Stanford University, July, 1974.

[Taylor 80] Taylor, R.N.
Assertions In Programming Languages.
Sigplan Notices 15(1), Jan, 1980.

[Teitelbaum 80] Teitelbaum, Tim and Reps, Thomas.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Technical Report TR 80-421, Cornell University, Department of Computer Science, May, 1980.

[Teitelman 77] Teitelman, Warren.
A Display Oriented Programmar's Assistant.
Technical Report CSL-77-3, Xerox Parc:Palo Alto, March, 1977.

[Teitelman 78] Teitelman, Warren.
Interlisp Reference Manual
XEROX Palo Alto Research Center, 1978.

[Thacker 79] Thacker, C.P., et.al.
Alto: A Personal Computer.
Technical Report CSL-79-11, Xerox Parc:Palo Alto, Aug, 1979.

[Tichy 80] Tichy, Walter.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, Jan, 1980.

[Unix 81a] *Unix Programmer's Manual*
Seventh Edition edition, 1981.

[Unix 81b] Katseff, H.P.
Sdb: A Symbolic Debugger
Unix User's Manual edition, 1981.

[VMS 78] *VAX VMS Manual*
Digital Equipment Corp., 1978.

[Warren 75] Warren, H.
Design of The FDS Interactive Debugging System.
IBM Report , 1975.

[Wilcox 76] Wilcox, T.R., et.al.
The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System.
Communications of the ACM 19(11), Nov, 1976.

[Wirth 77] Wirth, Niklaus.
Modula: A Language for Modular Programming.
Software - Practice and Experiance 7(1), 1977.

[Wulf 75] Wulf, William, et.al.
Programming Languages Series. Volume 2: *The Design of an Optimizing Compiler.*
American Elsevier Publishing Company Inc., 1975.

[Wulf 79] Wulf, W., Feiler, P.H., Brender R., Zinnikas, J.
A Quantitative Technique For Comparing the Quality of Language Implementations.
1979.
Working Paper, Computer Science, Carnegie-Mellon University.

[Wulf 80] Wulf, W.A.
PQCC: A Machine-Relative Compiler Technology.
Technical Report CMU-CS-80-144, Carnegie-Mellon University, Computer Science, Sept, 1980.

[Yarwood 77] Yarwood, E.
Toward Program Illustration.
Master's thesis, University of Toronto, Oct, 1977.